



RAPPORTI TECNICI INGV

Piattaforma Web per il Monitoraggio
e la gestione delle stazioni della Rete
Sismica Mobile dell'Osservatorio Etneo



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

522

**Direttore Responsabile**

Daniela VERSACE

Editor in Chief

Milena MORETTI

Editorial Board

Laura ALFONSI

Sofia BARANELLO

Carla BARNABA

Christian BIGNAMI

Simona CARANNANTE

Maria Grazia CIACCIO

Luca COCCHI

Luigi CUCCI

Lorenzo CUGLIARI

Elisabetta D'ANASTASIO

Alessia DI CAPRIO

Roberto DI MARTINO

Domenico DI MAURO

Domenico DORONZO

Filippo GRECO

Alessandro IAROCCI

Mario MATTIA

Daniele MELINI

Anna NARDI

Lucia NARDONE

Marco OLIVIERI

Arianna PESCI

Pierangelo ROMANO

Paola SBARRA

Davide SCAFIDI

Maurizio SOLDANI

Sara STOPPONI

Umberto TAMMARO

David ZULIANI

REGISTRAZIONE AL TRIBUNALE DI ROMA N.173 | 2014, 23 LUGLIO

© INGV Istituto Nazionale di Geofisica e Vulcanologia

Rappresentante legale: Presidente INGV

Sede: Via di Vigna Murata, 605 | Roma

Tutti i diritti riservati

www.ingv.it



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

RAPPORTI TECNICI INGV

Piattaforma Web per il Monitoraggio e la gestione delle stazioni della Rete Sismica Mobile dell'Osservatorio Etneo

Web Platform for Monitoring and Management of the Etna Observatory's Mobile Seismic Network Stations

Carmelo La Piana*, Fabrizio Pistagna, Mario Torrisi

INGV | Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania - Osservatorio Etneo

*Corresponding author

Accettato 21 aprile 2026 | Accepted 21 April 2026

Come citare | How to cite La Piana, C., Pistagna, F., Torrisi, M. (2026). Piattaforma Web per il Monitoraggio e la gestione delle stazioni della Rete Sismica Mobile dell'Osservatorio Etneo. *Rapporti Tecnici INGV*, 522, 46 pp. <https://doi.org/10.13127/rpt/522>

In copertina Immagine generata con Google Gemini (04/12/2025) su prompt dell'autore | Cover Image generated with Google Gemini (04/12/2025) at the author's prompt

522

INDICE

Riassunto	7
<i>Abstract</i>	7
Introduzione	8
1. Metodologia di progettazione e sviluppo	8
2. Architettura del sistema	9
2.1 Database (MS SQL Server)	9
2.2 Servizi in <i>background</i> (<i>IHostedService</i>)	10
2.3 Servizi di supporto (logica applicativa)	11
2.4 Autenticazione ibrida	12
2.5 <i>Frontend web</i> (<i>Razor Pages</i> e interattività <i>Client-Side</i>)	12
3. Implementazione dell’Interfaccia Utente	13
3.1 Pagina <i>Dashboard</i>	14
3.2 Pagina Dettaglio (analisi stazione)	16
3.3 Pagina riepilogo stati (Calendario affidabilità)	17
3.4. Pagina Centro di controllo	18
3.5 Gestione del flusso di lavoro degli interventi	19
4. Dettagli implementativi	20
4.1 Configurazione dei servizi in <i>background</i> (<i>IHostedService</i>)	20
4.2 Logica di autenticazione ibrida	21
4.3 Gestione della sicurezza e principio del minimo privilegio	22
4.4 <i>Data ingestion</i> : gestione robusta dei dati <i>CSV</i> (<i>data cleaning</i>)	23
4.5 Visualizzazione immagini “live” (<i>ImageStreamer</i>)	23
4.6 Logica di definizione dello stato giornaliero	24
4.7 Configurazione dell’ambiente di produzione (<i>IIS</i>)	24
4.8 Generazione <i>Report PDF</i> dinamici	25
4.9 Gestione dei dati <i>time-series</i> e stato giornaliero	25
5. Validazione e risoluzione delle criticità	26
5.1 Validazione dei servizi in <i>background</i> (<i>fault tolerance</i>)	27
5.2 Validazione della logica di stato (<i>race conditions</i>)	27
5.3 Gestione delle criticità in produzione: caso di studio (<i>Schema Drift</i>)	27
6. Conclusioni	28
7. Disponibilità del codice sorgente	28
Contributi degli autori	29

Specifiche tecniche e risorse software	29
Sitografia	29
Appendici	31

Riassunto

Questo rapporto tecnico presenta l'architettura, il flusso operativo e le scelte implementative alla base di una piattaforma *web* centralizzata, progettata per il monitoraggio in tempo reale dello stato di funzionamento della Rete Sismica Mobile gestita dall'Osservatorio Etneo. La piattaforma è stata sviluppata per superare le limitazioni intrinseche dei processi di supervisione manuale, introducendo un sistema automatizzato capace di integrare e analizzare in modo continuo flussi eterogenei di dati, tra cui *file CSV* di stato e immagini dei sismogrammi di dati giornalieri generati in automatico da *scripts Python*.

L'applicazione, sviluppata in *ASP.NET Core 8.1* seguendo il modello delle *Razor Pages*², costituisce un sistema di monitoraggio completamente automatizzato. Grazie all'impiego di servizi in *background* basati su *IHostedService*³, il sistema garantisce un controllo continuo dei parametri critici come la tensione delle batterie e la stabilità della connettività, e attiva automaticamente notifiche immediate in caso di qualsiasi anomalia rilevata.

L'interfaccia unificata, arricchita da una *dashboard* geospaziale interattiva e da moduli di analisi storica dei segnali, offre agli utenti una visione d'insieme immediata e ad alta densità informativa. L'automazione dei controlli riduce i tempi di rilevamento dei guasti e facilita la pianificazione degli interventi (gestiti nell'ambito della stessa interfaccia) limitando i periodi di inattività della rete.

Abstract

This technical report presents the architecture, operational flow, and implementation choices underlying a centralized web platform designed for real-time monitoring of the operating status of Mobile Seismic Network managed by the Etna Observatory. The platform was developed to overcome the inherent limitations of manual supervision processes by introducing an automated system capable of continuously integrating and analyzing heterogeneous data streams, including status CSV files and images of daily seismogram data automatically generated by Python scripts. The application, developed in ASP.NET Core 8 following the Razor Pages model, constitutes a fully automated monitoring system. Thanks to the use of background services based on IHostedService, the system ensures continuous monitoring of critical parameters such as battery voltage and connectivity stability, and automatically triggers immediate notifications in the event of any detected anomalies.

The unified interface, enhanced by an interactive geospatial dashboard and signal history analysis modules, provides users with an immediate and highly informative overview. Automated checks reduce fault detection times and facilitate intervention planning (managed within the same interface), limiting network downtime.

Keywords Monitoraggio real-time; ASP.NET Core; Stazioni Sismiche | Real-time monitoring; Seismic stations; Automatic alerts

¹ Framework open-source per lo sviluppo di applicazioni web ad alte prestazioni, basato su .NET.

² Tecnologia di ASP.NET Core per creare pagine web dinamiche con separazione tra logica e presentazione.

³ Interfaccia di ASP.NET Core che consente di implementare servizi in *background*, eseguiti in modo continuo e indipendente dal traffico web.

Introduzione

Il monitoraggio delle stazioni sismiche in aree remote non riguarda solo l'acquisizione del dato scientifico, ma la sorveglianza dell'intero "*State of Health*" (SoH) della catena strumentale: dai sensori ai digitalizzatori, fino ai sistemi di alimentazione e trasmissione. Gestire flussi di dati così eterogenei e verificare che ogni componente sia operativo richiede, in assenza di una piattaforma centralizzata, controlli manuali lunghi e complessi, che inevitabilmente rallentano i tempi di intervento in caso di guasto. Il sistema proposto nasce proprio per colmare questa lacuna, garantendo un controllo capillare anche su quelle stazioni temporanee, spesso usate per progetti di ricerca o analisi del *noise* ambientale, che solitamente sfuggono ai sistemi di monitoraggio *standard*. L'obiettivo è stato realizzare un unico ambiente *web* che agisca sia da *dashboard* di controllo in tempo reale che da archivio storico. L'applicazione è progettata per acquisire dati grezzi da *share* di rete protette, normalizzare formati non *standard* e monitorare costantemente i parametri vitali (tensione batterie, temperature, stato *GPS*), offrendo al contempo un'interfaccia intuitiva per l'analisi visiva dei sismogrammi e la gestione delle manutenzioni.

Dal punto di vista architetturale, la scelta di sviluppare la piattaforma su *ASP.NET Core 8* [1], discostandosi da *framework Python* come *Django* o *FastAPI*, che pure rappresentano uno standard nel mondo scientifico, risponde a precise necessità infrastrutturali. Dovendo integrare il sistema in un ambiente preesistente basato su server *Windows* e *IIS* (*Internet Information Services*), lo *stack Microsoft* ci ha garantito un *deployment* più naturale e sicuro rispetto alle alternative. Inoltre, l'utilizzo di un linguaggio fortemente tipizzato come *C#* offre una robustezza superiore nella gestione di processi concorrenti e a lungo termine, essenziale per un servizio che deve restare sempre attivo. In questa ottica ibrida, *Python* non viene abbandonato, ma viene impiegato specificamente "dietro le quinte" per gli script di analisi dati, lasciando a *.NET* il compito di gestire l'orchestrazione *web*, la sicurezza e le prestazioni generali dell'applicazione.

1. Metodologia di progettazione e sviluppo

La progettazione della piattaforma ha seguito un approccio metodologico concordato con il personale della Rete Sismica Mobile (RSM) dell'Osservatorio Etneo (OE). La necessità principale era quella di convertire un insieme di procedure di controllo manuali, temporalmente discontinue e disomogenee, in un flusso operativo coerente, automatizzato e affidabile.

Durante la fase preliminare di analisi sono stati identificati diversi punti critici operativi (*pain points*), tra cui la difficoltà nel correlare in modo rapido e consistente i dati relativi allo *State of Health* (SoH) (trasmessi come *file* di testo contenenti le telemetrie di tensione e funzionamento) con i corrispondenti sismogrammi, oltre alla latenza nel rilevamento dei guasti di alimentazione. A partire da queste esigenze, lo sviluppo ha seguito un ciclo iterativo e incrementale: inizialmente è stata consolidata la *pipeline* di acquisizione dei dati grezzi attraverso servizi in *background*, assicurando la continuità del *polling* e la validazione dei dati. Successivamente, la logica di classificazione dello stato operativo delle stazioni (es. "*Online*" o "*Intermittente*") è stata raffinata sulla base di criteri dinamici e soglie configurabili.

In parallelo, l'interfaccia utente è stata progressivamente arricchita con componenti fondamentali come la mappa interattiva, indicatori visivi di allerta e strumenti che consentono una correlazione immediata tra fonti eterogenee di informazione. Questo processo iterativo ha permesso di ottenere un sistema stabile, il cui codice risulta altamente leggibile e facilmente integrabile nel flusso di lavoro quotidiano per la gestione della RSM.

2. Architettura del sistema

Il sistema è progettato su un'architettura a tre livelli (come illustrato in Figura 1) pensata per garantire scalabilità, sicurezza e manutenibilità, un *database SQL Server* per la persistenza, un *backend ASP.NET Core* per la logica di *business* e l'elaborazione, e un *frontend Razor Pages* per l'interfaccia utente (come illustrato in Figura 2).

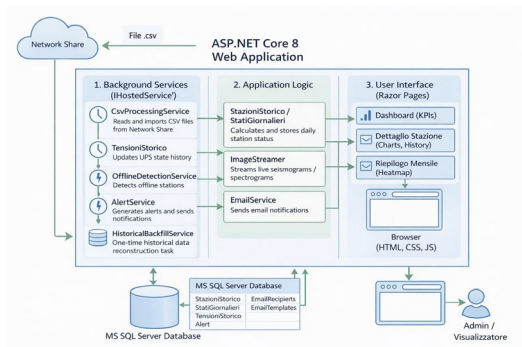


Figura 1 Schema generale dell'applicazione che illustra l'interazione tra i tre blocchi principali, i servizi di *backend* in esecuzione continua (*IHostedService*), la logica applicativa e l'interfaccia utente, con evidenza del flusso dati dalla *share* di rete al *database*.

Figure 1 General application diagram illustrating the interaction between the three main blocks: the continuously running backend services (*IHostedService*), the application logic and the user interface, highlighting the data flow from the network share to the database.

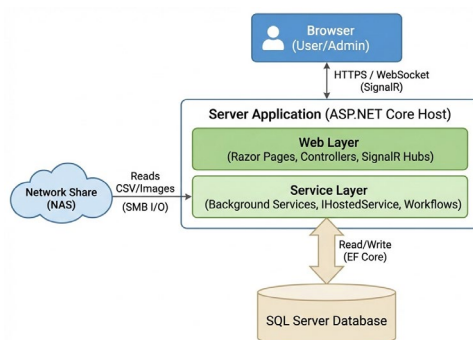


Figura 2 Diagramma che mostra la separazione tra *Frontend*, *Backend* (*ASP.NET Core*) e livello dati (*SQL Server* gestito tramite *Entity Framework Core*).

Figure 2 Diagram showing the separation between *Frontend*, *Backend* (*ASP.NET Core*) and data tier (*SQL Server* managed via *Entity Framework Core*).

2.1 Database (MS SQL Server)

Gestito tramite *Entity Framework Core*⁴, il *database* rappresenta il cuore della persistenza del sistema. Lo schema relazionale è stato progettato per bilanciare l'efficienza nelle operazioni in tempo reale con la necessità di storicizzare grandi volumi di dati, garantendo sempre l'integrità delle informazioni.

⁴ *Entity Framework Core* (EF Core) ORM (Object-Relational Mapper) per .NET che semplifica l'interazione con *database* relazionali tramite oggetti C#.

Le principali entità del modello dati includono (come illustrato in Figura 3):

- **StazioniSismiche:** è il registro centrale dell'applicazione. Contiene l'anagrafica completa e lo stato corrente di ogni stazione, inclusi i parametri telemetrici, gli stati operativi (*flag IsOffline*, *InManutenzione*) e le coordinate geospaziali per la mappa.
- **TensioniStorico:** tabella dedicata alla memorizzazione delle serie temporali dei dati di tensione. È la fonte dati principale per la generazione dei grafici di analisi storica e per il monitoraggio dei *trend* di consumo energetico.
- **Interventi:** funge da diario operativo completo (*audit trail*). Traccia ogni attività di manutenzione, registrando l'autore, la data di apertura/chiusura e la motivazione tecnica, garantendo la tracciabilità del ciclo di vita dei guasti.
- **StatiGiornalieri:** per evitare l'aggregazione in tempo reale di milioni di record grezzi, la tabella memorizza il pre-calcolo dello stato giornaliero. In questo modo, la visualizzazione dei calendari annuali richiede la lettura di poche decine di righe invece di milioni, riducendo drasticamente il carico sul database e l'impatto sull'occupazione di spazio su disco è trascurabile.
- **Configurazione e utenti:**
 - *AspNetUsers:* gestisce le identità, i ruoli e i permessi degli utenti.
 - *GlobalSettings:* centralizza i parametri globali di funzionamento del sistema (es. soglie di voltaggio per gli allarmi, *timeout* per l'*off-line*). Permette agli amministratori di tarare il comportamento dell'algoritmo a *runtime* (senza necessità di ricompilare o riavviare il servizio), applicando le modifiche istantaneamente a tutte le stazioni monitorate.
 - *EmailTemplates:* contiene i *layout HTML* per le notifiche, rendendo i messaggi personalizzabili direttamente dall'interfaccia di amministrazione senza ricompilare il codice.

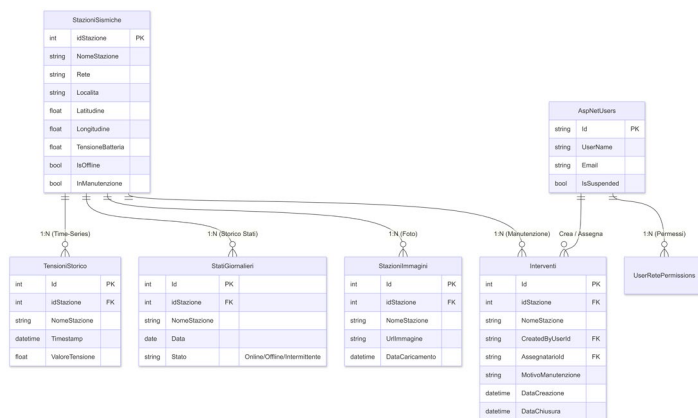


Figura 3 Diagramma Entità-Relazione del database che evidenzia le entità principali e le relazioni logiche progettate per garantire l'integrità referenziale e l'efficienza delle query.

Figure 3 Database Entity-Relationship Diagram highlighting the primary entities and logical relationships designed to ensure referential integrity and query efficiency.

2.2 Servizi in *background* (*IHostedService*)

Il nucleo operativo dell'applicazione è costituito dai servizi in *background* che operano in modalità continua, garantendo l'elaborazione automatica dei dati e l'identificazione tempestiva delle criticità. L'architettura adotta un paradigma di esecuzione fortemente asincrono, l'implementazione tramite l'interfaccia *IHostedService* di *ASP.NET Core* permette di disaccoppiare nettamente la logica di elaborazione dati dal ciclo di vita delle richieste *HTTP*. Questo approccio

rende il sistema non bloccante, le operazioni complesse come il *parsing* dei *file* o l'analisi storica avvengono su *thread* separati, lasciando l'interfaccia *web* sempre reattiva per l'operatore, anche durante i picchi di carico (come illustrato nel *workflow* in Figura 4). I principali componenti del sistema sono qui elencati:

- **CsvProcessingService** (Il motore che legge i dati):
agisce come *data ingestion engine*, responsabile dell'acquisizione sicura dei dati grezzi dalla *share* di rete e della loro normalizzazione nel *database*.
- **OfflineDetectionService** (Il modulo di monitoraggio):
funge da monitor di disponibilità; analizza la latenza dei pacchetti dati ricevuti per determinare in tempo reale lo stato operativo (*Online/Offline*) di ogni stazione, segnalando le interruzioni di trasmissione non appena superano le soglie di tolleranza definite.
- **AlertService** (Il modulo per la gestione delle notifiche):
è il motore di *event dispatching* che dopo aver analizzato le telemetrie (es. tensione batteria) genera messaggi e comunicazioni verso il personale.
- **HistoricalBackfillService** (Analisi Storico):
un componente di manutenzione dati che garantisce la consistenza delle serie storiche pregresse, essenziale per la generazione delle statistiche di affidabilità.

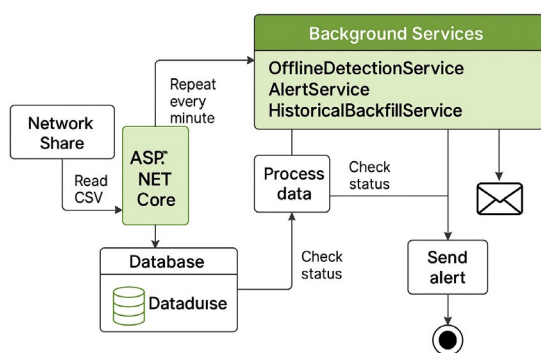


Figura 4 Ciclo di vita dei servizi in *background*. Diagramma di flusso che descrive anche le tempistiche e le responsabilità dei servizi *CsvProcessingService*, *OfflineDetectionService* e *AlertService*.

Figure 4 Background services lifecycle. Flowchart detailing the timing and responsibilities of the *CsvProcessingService*, *OfflineDetectionService*, and *AlertService* services.

2.3 Servizi di supporto (logica applicativa)

Oltre ai servizi di *background* che costituiscono il “cuore” del sistema, l'architettura integra componenti di supporto trasversali, essenziali per orchestrare i flussi di comunicazione. L'utilizzo di questi moduli condivisi migliora significativamente la manutenibilità del codice e applicando il principio *DRY* (*Don't Repeat Yourself*), centralizzando la logica comune, si evita la duplicazione di codice e si riduce il rischio di errori in fase di aggiornamento (come illustrato in Figura 5).

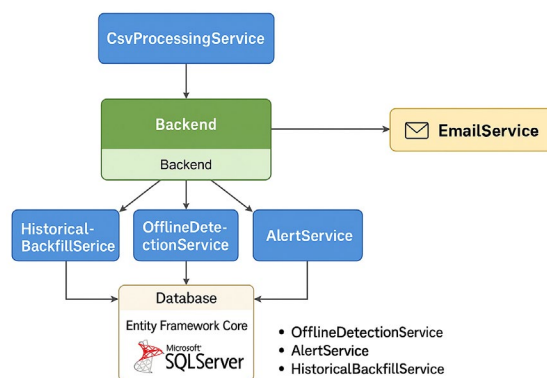
Tra questi, l'*EmailService* riveste un ruolo importante. Sebbene non sia un servizio autonomo (*IHostedService*), agisce come un modulo infrastrutturale invocato dai servizi di monitoraggio (*AlertService*, *OfflineDetectionService*) per gestire l'intero ciclo di vita delle notifiche.

Le sue funzioni principali sono state progettate per garantire la massima flessibilità:

- **Template Dinamici**: il servizio non utilizza testo nel codice, ma recupera i *layout HTML* direttamente dalla tabella *EmailTemplates* all'interno del *database*. Questo permette di modificare la grafica o il testo delle *e-mail*.
- **Rendering dei Segnaposto**: prima dell'invio, il motore di *rendering* sostituisce i segnaposto dinamici (es. {NomeStazione}, {DataEvento}) con i dati reali dell'evento rilevato, generando messaggi personalizzati e contestuali.
- **Invio Centralizzato**: le *e-mail* vengono instradate tramite il server *SMTP* configurato a livello globale.

Figura 5 Schema delle dipendenze che mostra come *EmailService* agisca da modulo trasversale, invocato dai diversi servizi di *background* per la gestione centralizzata delle comunicazioni via SMTP.

Figure 5 Dependency diagram showing how *EmailService* acts as a cross-platform module, invoked by various background services for centralized management of SMTP communications.



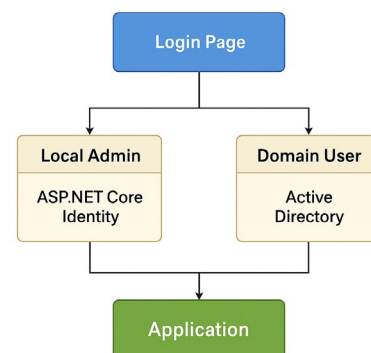
2.4 Autenticazione ibrida

Per garantire la continuità operativa ed evitare che l'indisponibilità del *controller* di dominio⁵ blocchi l'accesso al sistema (*Single Point of Failure*), è stata progettata un'architettura di autenticazione ibrida. Questo *pattern* consente di unificare l'esperienza utente attraverso una singola interfaccia di *login*, instradando le richieste verso due *Identity Provider* distinti in base alla tipologia di *account*:

- **Provider Istituzionale (Active Directory):** gestisce l'autenticazione del personale operativo, delegando la validazione delle credenziali all'infrastruttura di rete esistente. Questo garantisce il rispetto delle *policy* di sicurezza dell'istituto (nessuna duplicazione di *password*).
- **Provider Locale (Failover):** un meccanismo di *backup* dedicato esclusivamente all'amministratore di sistema. Questo canale parallelo è completamente disaccoppiato dalla rete e garantisce l'accesso alla *console* di gestione anche in scenari di isolamento o guasto dei servizi di *directory*.

Figura 6 Flusso di autenticazione ibrida. Diagramma che illustra la doppia strategia di accesso, validazione tramite *Active Directory* per gli utenti istituzionali e autenticazione locale per l'amministratore di sistema.

Figure 6 Hybrid authentication flow. Diagram illustrating the dual access strategy: *Active Directory* validation for institutional users and local authentication for the system administrator.



2.5 Frontend web (Razor Pages e interattività Client-Side)

Il livello di presentazione è stato realizzato adottando il pattern *Razor Pages*, una funzionalità nativa di *ASP.NET Core* che favorisce un'organizzazione del codice incentrata sulla pagina, piuttosto che sul *controller*. Questa scelta architetturale offre due vantaggi strategici:

- **Server-Side Rendering:** l'*HTML* viene generato dinamicamente sul *server*, garantendo un caricamento rapido iniziale e un accesso diretto e sicuro ai dati del *backend* senza esporre *API* pubbliche non necessarie.

⁵ Il *domain Controller* è il server che funge da "autorità centrale" per la sicurezza all'interno di una rete Windows. Gestisce l'elenco di tutti gli utenti e i computer autorizzati e ha il compito di verificare le credenziali e decidere chi può accedere a quali risorse.

- **Separazione delle Responsabilità:** ogni pagina è accoppiata a una classe C# dedicata che gestisce la logica di presentazione, mantenendo il *markup* pulito.

Per superare la staticità tipica delle pagine renderizzate lato server e permettere un monitoraggio in tempo reale, il *frontend* è stato integrato con componenti lato *client* basati su librerie *JavaScript*. L'aggiornamento delle informazioni avviene in modo asincrono sfruttando la *Fetch API*⁶ e la libreria *SignalR7* per la comunicazione bidirezionale. Questo *stack* tecnologico permette di interrogare i servizi di *backend* e manipolare dinamicamente il *DOM* 8(*Document Object Model*), aggiornando solo i componenti necessari della pagina senza richiederne il ricaricamento completo.

3. Implementazione dell'Interfaccia Utente

L'interfaccia è stata progettata utilizzando *Bootstrap 59* [7] seguendo un rigoroso approccio *mobile-first*, per garantire la piena compatibilità con l'intero spettro di dispositivi in uso, dai *desktop* delle sale operative ai *tablet* e *smartphone* utilizzati sul campo (vedi Figura 8). L'obiettivo primario è offrire un'esperienza utente intuitiva, veloce e coerente, minimizzando la curva di apprendimento per gli operatori.

Le caratteristiche architetture principali includono:

- **Design Responsive e Navigazione Gerarchica:** il *layout* utilizza griglie adattive e componenti ottimizzati per il *touch*. La navigazione è strutturata tramite *tabs* e *card*, organizzando le informazioni in modo gerarchico per facilitare la consultazione rapida.
- **Interattività Avanzata:** i grafici, realizzati con la libreria *ApexCharts*¹⁰[4], supportano lo *zoom XY* e visualizzano soglie di allarme dinamiche. L'aggiornamento dei dati avviene in tempo reale tramite *AJAX* e *SignalR11*, eliminando la necessità di *refresh* completi della pagina e garantendo la percezione di un sistema "vivo".
- **Feedback Immediato:** le operazioni critiche (es. cambio stato stazione) sono confermate da notifiche "*Toast*" non intrusive, che informano l'utente sull'esito dell'azione mantenendolo nel contesto operativo.
- **Accessibilità:** la palette cromatica e i contrasti sono stati selezionati in conformità agli *standard WCAG 2.1 12* (Livello AA) per assicurare la leggibilità in ogni condizione di luce.

L'applicazione è organizzata in quattro macro-sezioni (come illustrato in Figura 7).

1. **Dashboard:** sala di controllo con *KPI* (vedi paragrafo 3.1), mappa geospaziale e filtri rapidi.
2. **Pagina Dettaglio:** *hub* di analisi con grafici storici, sismogrammi e immagini *live*.
3. **Calendario affidabilità:** vista *heatmap* per l'analisi dei *pattern* di funzionamento mensili.
4. **Centro di controllo:** area amministrativa per la gestione di utenti, reti e configurazioni.

⁶ Standard per chiamate *AJAX* basate su *Promise*

⁷ Libreria *open-source* per *ASP.NET Core* che facilita l'implementazione di funzionalità web in tempo reale.

⁸ Interfaccia di programmazione standardizzata dal W3C che rappresenta la pagina web come una struttura ad albero di oggetti.

⁹ *Framework CSS* per la creazione di interfacce responsive e *mobile-first*.

¹⁰ Libreria *JavaScript* per la creazione di grafici interattivi e dinamici.

¹¹ Libreria di *ASP.NET Core* per comunicazione *real-time* tra server e client via *WebSocket*.

¹² Le *WCAG (Web Content Accessibility Guidelines)* sono le linee guida internazionali sviluppate dal W3C per rendere i contenuti web accessibili a un più ampio numero di persone.

Figura 7 Wireframe della struttura di navigazione. Progettazione dell'interfaccia utente, organizzata gerarchicamente in *dashboard*, dettaglio stazione, calendario stati e area Admin.

Figure 7 Navigation structure wireframe. User interface design, hierarchically organized into dashboard, station details, status calendar, and Admin area.

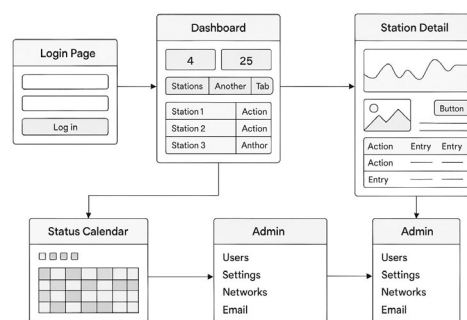
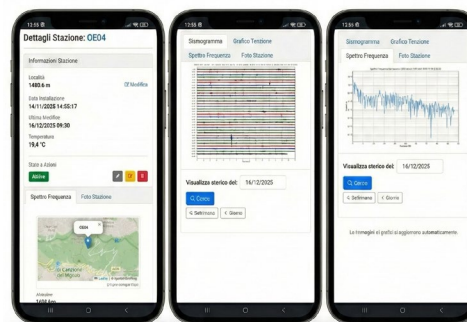


Figura 8 Mockup di visualizzazione dell'interfaccia su *smartphone*, a dimostrazione dell'approccio *mobile-first*. Le schermate mostrano, da sinistra a destra: i dettagli e lo stato della stazione, il sismogramma e lo spettro di frequenza.

Figure 8 Smartphone interface mockup, demonstrating the mobile-first approach. From left to right, the screenshots show the station details and status, the multichannel seismogram, and the frequency spectrum.



3.1 Pagina *Dashboard*

La *Dashboard* rappresenta il pannello di controllo principale del sistema, progettato per offrire una panoramica immediata dello stato della rete e permettere agli operatori di intervenire rapidamente (come illustrato in Figura 9 e 10).

L'interfaccia è organizzata in sezioni funzionali distinte:

- **KPI (Key Performance Indicators)**

Nella parte superiore, quattro riquadri mostrano il conteggio delle stazioni suddivise per stato: “Attiva”, “Offline”, “In Manutenzione” e “Allarme Batteria”. La logica di calcolo assicura che i conteggi siano mutuamente esclusivi, evitando duplicazioni e fornendo un quadro chiaro della situazione complessiva.

- **Mappa Interattiva**

Subito sotto i KPI, la mappa realizzata con le librerie *Leaflet.js*¹³ [6] offre una rappresentazione geospaziale della rete. Il caricamento dei dati avviene tramite il metodo *OnGetAsync* (in *Index.cshtml.cs*), che interroga tutte le stazioni attive con coordinate valide. Le informazioni di stato (es. *IsOffline*, *TensioneBatteria*) vengono serializzate in un oggetto *JSON* (*StazioniPerMappaJson*) e passate al *client*. Uno script *JavaScript* legge questi dati e popola la mappa, colorando i *marker* dinamicamente in base allo stato operativo:

- **Verde:** “Stazione attiva”.
- **Rosso:** “Stazione *offline*”.
- **Giallo:** “Stazione in manutenzione”.
- **Arancione:** “Batteria scarica”.

¹³ Libreria *JavaScript* leggera per la visualizzazione di mappe interattive.

L’interattività è garantita cliccando su un *marker*, che apre un *popup* con i dati essenziali (nome, tensione) e un *link* diretto alla pagina di dettaglio.

Tabelle di Stato e Feedback

La *dashboard* include tabelle organizzate in schede (*tabs*) per ciascuno stato, responsive e filtrabili. Infine, per garantire un’esperienza utente fluida, le operazioni amministrative (es. “Metti in Manutenzione”) sono gestite in modo asincrono. Il sistema fornisce un *feedback* immediato tramite notifiche “*Toast*”¹⁴ non invasive, confermando il successo dell’azione senza richiedere il ricaricamento della pagina.



Figura 9 Dashboard operativa. La schermata principale che integra i KPI di stato (“Stazioni Attive”, “Offline”, “In Manutenzione”) e la mappa geospaziale interattiva basata su *Leaflet.js* per la localizzazione immediata delle anomalie.

Figure 9 Operational dashboard. The main screen integrates status KPIs (“Active Station”, “Offline”, “Under Maintenance”) and an interactive geospatial map based on *Leaflet.js* for immediate location of anomalies.

Attive		Offline	In Manutenzione	Disomogeneo		
Nome Stazione	Rete/Progetto	Azioni				
GERN1 - Serra la Nave - sbarco	YK					
GERN3 - Serra la Nave - radice	YK					
INARV3 - Pozzo Archeologico Kametina	IV					
INARV5 - Pozzello - Protezione Civile	IV					
INARV8 - Lidoia - Ufficio Tecnico Comunale	IV					
INARV7 - Riserva Pantalica - Anasap	IV					
MUSI1 - Tramezzelle - Cinisera	TC					
MUSI2 - Padana - Scuola Canale	TC					
MUSI3 - S. Giovanni la Punta - DPS Regionale	TC					
ODE1 - S. Anna Il Battenti - Motorizzazione Civile	YK					
ODE2 - Marescaucia - Scuola C. Marchini	YK					
ODE3 - Zafferana S. - S. da Cassone	YK					
ODE4 - Belvedere - Casa Codina	N/A					
ODE5 - Santa Venera al Pozzo	N/A					
ODE6 - S. Gregorio di Catania - Immacolata e M. Carlo	N/A					
ODE4 - Vulcano - Villaggio Lento	N/A					
ODE7 - Lipari - M. Rosa	N/A					
ODE8 - Nicolosi - Sede INGV	YK					

Figura 10 Lista tabellare delle stazioni. Vista riepilogativa che offre strumenti di gestione rapida per ogni stazione, inclusi i comandi per la generazione di report *PDF*, la messa in manutenzione e l’accesso ai dettagli.

Figure 10 Tabular list of stations. Summary view that offers quick management tools for each station, including commands for generating *PDF* reports, maintenance commissioning, and accessing details.

¹⁴ Elementi dell’interfaccia utente (UI) leggeri e non invasivi che appaiono temporaneamente, solitamente in un angolo dello schermo, per fornire *feedback* su un’operazione.

3.2 Pagina Dettaglio (analisi stazione)

La pagina di dettaglio rappresenta la sezione più complessa dell'interfaccia e offre una vista completa sullo stato e sui dati di ciascuna stazione. È progettata per combinare informazioni statiche, grafici interattivi e contenuti multimediali, garantendo un'esperienza fluida anche su dispositivi mobili (come illustrato in Figura 11).

Il layout è strutturato su due colonne grazie a *Bootstrap 5*, utilizzando la classe *d-flex align-items-stretch*¹⁵ per mantenere un'uniformità visiva perfetta tra il riquadro delle informazioni e quello dei grafici.

La pagina è suddivisa in due macro-sezioni funzionali.

Informazioni stazione (colonna di sinistra)

Questo riquadro raccoglie i dati anagrafici e di stato (es. "Tensione", "Rete", "Coordinate GPS"). È stata implementata una logica di visualizzazione difensiva che gestisce i valori nulli, mostrando "N/D" per le stazioni che non dispongono ancora di tutti i sensori. Questo approccio garantisce la retrocompatibilità con i *record* esistenti e supporta l'inserimento di nuove stazioni anche con dati parziali.

Area di analisi multimediale (colonna destra)

Questa sezione utilizza un sistema di navigazione a schede (*Tabs*) per organizzare contenuti eterogenei senza occupare eccessivo spazio verticale. L'operatore può commutare istantaneamente tra quattro viste principali:

- **Sismogramma (Tab 1):** questo grafico rappresenta il primo livello di diagnostica *hardware*. La visualizzazione della traccia sismica permette di valutare immediatamente l'operatività del sensore, identificando anomalie macroscopiche come il *drift* del segnale, la presenza di *spike* elettronici o l'assenza totale di *input*. La possibilità di scorrere lo storico delle tracce è decisiva per individuare l'esatto momento di insorgenza di un guasto (vedi Figura 11).
- **Grafico Tensione (Tab 2):** realizzato con *ApexCharts*¹⁶, visualizza l'andamento della tensione negli ultimi sette giorni (vedi Figura 12). Include lo *Zoom XY* e una linea di soglia dinamica (rossa) letta dai *GlobalSettings*, fornendo un riscontro immediato su eventuali cali di potenza (vedi figura 12).
- **Spettro di frequenza (Tab 3):** strumento avanzato per l'analisi del contenuto spettrale del segnale. A differenza del sismogramma, questa vista permette all'operatore di selezionare e analizzare intervalli temporali specifici per discriminare la natura di un disturbo. Attraverso lo studio dello spettro di frequenza, è possibile distinguere tra rumore antropico/ambientale e disturbi strumentali interni, facilitando una diagnosi precisa della causa del problema senza necessità di recarsi in sito.
- **Foto Stazione (Tab 4):** la scheda "Foto Stazione" è stata progettata per supportare l'operatività sul campo, consentendo di documentare visivamente lo stato del sito e gli interventi eseguiti. L'architettura del modulo è stata ottimizzata per i dispositivi mobili e per garantire la sicurezza dei dati.
- **Upload Mobile-First:** il *form* di caricamento (configurato come *multipart/form-data*) utilizza l'attributo *accept="image/*"*, che sui dispositivi mobili abilita l'acquisizione diretta dalla fotocamera, semplificando il lavoro degli operatori.
- **Sicurezza dello Storage:** per garantire l'isolamento dei dati, le immagini non vengono

¹⁵ Classi di utilità di *Bootstrap* che implementano il layout *Flexbox*, obbligando tutti gli elementi interni a espandersi verticalmente in modo da riempire completamente l'altezza del contenitore genitore.

¹⁶ Libreria *JavaScript open-source* per la visualizzazione dati. È stata selezionata per il suo supporto nativo alle *time-series* e per le funzionalità di interattività avanzata lato *client*.

salvate nella cartella pubblica `wwwroot`, ma in un percorso protetto sul server (`..\StazioneImmagine\[NomeStazione]`).

- **Accesso via Streaming:** l'accesso diretto ai *files* via *IIS* è bloccato. Le immagini vengono servite esclusivamente tramite un *handler* dedicato (`OnGetStationImageAsync`) che legge il *file* dal percorso fisico e lo trasmette in *streaming* al *browser*. La visualizzazione finale è gestita dalla libreria **Fancybox** [8], che offre una galleria interattiva e intuitiva.

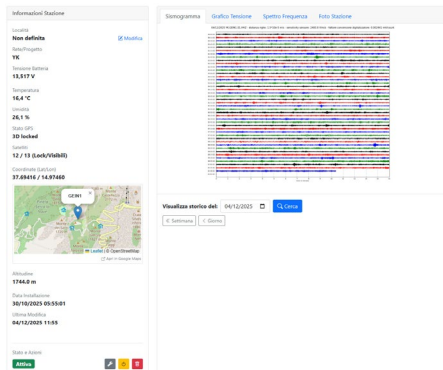


Figura 11 Pagina dettaglio stazione, dove sono visibili: il *box* con le informazioni sulla stazione, tipo geolocalizzazione su mappa con coordinate, temperatura, etc. nelle altre *tabs* ci sono i sismogrammi, gli spettri o il grafico della tensione.

Figure 11 Station detail page, where the following elements are displayed: an information box containing station data, such as map-based geolocation with coordinates, temperature, etc. In the other tabs, users can view the seismograms, the spectral analyses, or the battery voltage chart.

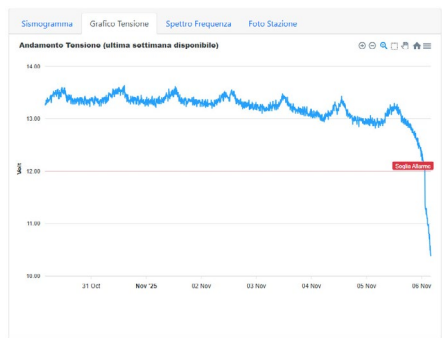


Figura 12 Grafico interattivo della tensione, realizzato con *ApexCharts* che mostra il trend di scarica della batteria negli ultimi 7 giorni, evidenziando la soglia critica di allarme (linea rossa).

Figure 12 Interactive voltage graph, created with ApexCharts, showing the battery discharge trend over the last 7 days, highlighting the critical alarm threshold (red line).

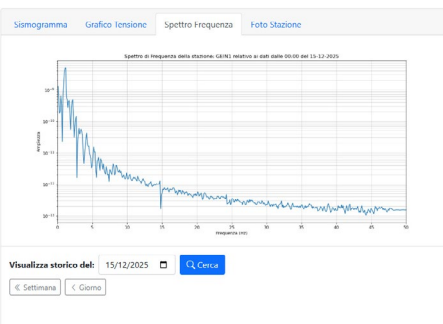


Figura 13 Visualizzazione dello spettro di frequenza generato in tempo reale. Il sistema di navigazione a schede permette di alternare tra sismogrammi, spettrogrammi e dati di tensione senza ricaricare la pagina.

Figure 13 View of the frequency spectrum generated in real time. The tabbed navigation system allows you to switch between seismograms, spectrograms, and voltage data without reloading the page.

3.3 Pagina riepilogo stati (Calendario affidabilità)

Questa pagina offre una panoramica completa dello stato di funzionamento della rete, permettendo agli utenti di analizzare rapidamente lo stato storico delle singole stazioni (come illustrato in Figura 14).

Per garantire prestazioni elevate, i dati mostrati non vengono calcolati al momento della richiesta, ma provengono dalla tabella *StatiGiornalieri*, che viene popolata a priori dai servizi in *background* per il mese e l'anno selezionati. Questo approccio architetturale ottimizza il recupero delle informazioni, riducendo i tempi di caricamento ed evitando l'esecuzione di *query* complesse su grandi volumi di dati storici.

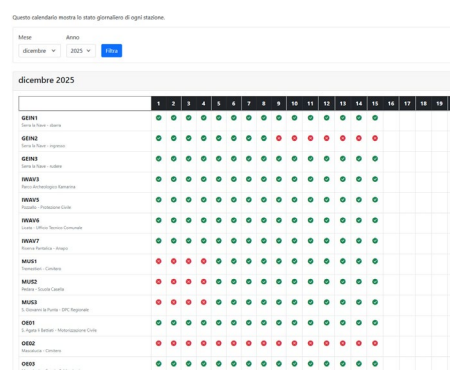
Il cuore della pagina è una tabella interattiva, dove ogni cella rappresenta lo stato di una stazione in uno specifico giorno. La codifica cromatica permette di individuare immediatamente *pattern* di funzionamento o anomalie ricorrenti.

- **Verde (Online):** la stazione ha trasmesso regolarmente.
- **Rosso (Offline):** nessun dato ricevuto per l'intera giornata.
- **Giallo (In Manutenzione):** la stazione era sottoposta a intervento tecnico.
- **Blu (Intermittente):** è stato rilevato un passaggio di stato (es. da *online* a *offline* o viceversa) durante le 24 ore.

Una legenda chiara accompagna il grafico, garantendo la comprensione anche agli utenti meno esperti. Grazie a questa struttura, la pagina supporta efficacemente le decisioni strategiche per gli interventi di manutenzione preventiva.

Figura 14 Calendario di affidabilità (*Heatmap*). Matrice visiva dello stato operativo mensile basata sui dati pre-calcolati. Il codice colore (Verde/Online, Rosso/Offline, Giallo/Manutenzione Blu/Intermittente) permette l'identificazione immediata dei pattern di guasto.

Figure 14 Reliability calendar (*Heatmap*). Visual matrix of monthly operational status based on pre-calculated data. Color coding (Green/Online, Red/Offline, Yellow/Maintenance, Blue/Intermittent) allows for immediate identification of failure patterns.



3.4. Pagina Centro di controllo

Per garantire al sistema un elevato livello di flessibilità e manutenibilità, è stata realizzata un'area di amministrazione accessibile esclusivamente agli utenti con ruolo *Admin* (come illustrato in Figura 15). Questa sezione funge da punto di controllo centralizzato, consentendo di gestire tutte le configurazioni operative senza intervenire direttamente sul *database*. Questo approccio limita il rischio di errore umano e garantisce l'integrità dei dati, poiché il *software* applica regole di validazione preventive (es. controllo dei *range* di valori o vincoli di chiave esterna) che impediscono l'inserimento di valori errati. L'area amministrativa è articolata in quattro moduli principali:

- **Gestione Utenti:** è il modulo dedicato alla sicurezza e agli accessi. Qui è possibile consultare l'elenco completo degli *account* abilitati ed eseguire operazioni rapide (es. sospensione, eliminazione). Per una gestione granulare, la pagina avanzata (*ManageUser*) permette di assegnare i ruoli tramite *ASP.NET Core Identity* e di configurare i permessi di rete. Agli utenti non amministratori può essere limitato l'accesso a specifiche reti (es. "YK" o "IV"), garantendo una rigorosa suddivisione.
- **Gestione Reti:** questo modulo offre le operazioni *CRUD*¹⁷ necessarie per creare e mantenere le anagrafiche di reti e progetti. È un componente infrastrutturale fondamentale, poiché l'intero sistema di autorizzazione dei Visualizzatori si basa sulla corretta mappatura logica tra le reti sismiche e gli utenti abilitati.
- **Configurazione Operativa:** questa sezione rappresenta la "sala macchine" del monitoraggio.

¹⁷ *CRUD*: è l'acronimo di *Create* (Creare), *Read* (Leggere), *Update* (Aggiornare), *Delete* (Cancellare). Rappresenta le quattro operazioni fondamentali per la gestione e la manipolazione dei dati nei *database* e nelle applicazioni *software*.

Consente di modificare le soglie critiche (es. Soglia Allarme Batteria) e i parametri temporali (es. Intervallo Stazione *Offline*) a caldo, senza dover riavviare l'applicazione. I valori vengono salvati in *GlobalSettings* e letti dinamicamente dai servizi in *background* (come *AlertService*) al ciclo di esecuzione successivo, rendendo le modifiche effettive in tempo reale.

- **Template E-mail:** questo modulo permette di gestire dinamicamente il contenuto delle notifiche automatiche. I *template HTML* sono memorizzati nel *database* e supportano l'uso di segnaposto (es. {NomeStazione}), che vengono sostituiti contestualmente dall'*EmailService* al momento dell'invio, permettendo di aggiornare il testo degli avvisi dinamicamente.

Centro di Controllo

Gestisci tutti gli aspetti dell'applicazione da un unico posto.

Gestione Utenti Gestione Reti Configurazione Template Email			
Elenco Utenti			
Cerca...		Cerca	Pulisci
Nome Utente	Email	Ruoli	Azioni
admin	admin@example.com	Admin	  
ctvalparone	salvatore.alparone@ingv.it	Admin	  
ctlapiana	carmelo.lapiana@ingv.it	Admin	  
ctlarocca	graziano.larocca@ingv.it	Admin	  

Figura 15 Centro di controllo dell'applicazione. Interfaccia di amministrazione centralizzata per la gestione degli utenti, la configurazione delle reti sismiche e la modifica “a caldo” dei parametri operativi e dei *template email*.

Figure 15 Application control center. Centralized administration interface for user management, seismic network configuration, and on-the-fly modification of operating parameters and email templates.

3.5 Gestione del flusso di lavoro degli interventi

La procedura di intervento viene avviata dalla “Pagina di Dettaglio” e gli operatori autorizzati possono dare inizio alla manutenzione selezionando l'apposito comando “Inizia Manutenzione”. L'attivazione di questo stato è immediatamente riscontrabile a livello visivo: l'interfaccia grafica segnala la condizione applicando un badge di colore giallo e un'icona a forma di chiave inglese sia nella *Dashboard* geospaziale che nelle tabelle di riepilogo.

Da un punto di vista logico, porre una stazione “In Manutenzione” non è solo un'etichetta grafica, ma un'azione che modifica il comportamento del *backend*: il sistema esclude temporaneamente la stazione dalle logiche di allarme automatico (monitoraggio *offline* e batteria), prevenendo l'invio di falsi positivi durante le operazioni tecniche sul campo.

La piattaforma, dunque, non si limita al monitoraggio passivo, ma integra un sistema completo per la gestione del ciclo di vita degli interventi, garantendo tracciabilità completa (*audit trail*) e continuità operativa. Questa funzionalità si basa sulla tabella “Interventi” del *database*, che funge da registro storico centralizzato.

Il flusso operativo segue un percorso strutturato in quattro fasi:

1. **Creazione dell'Intervento:** il processo può essere avviato da qualunque utente autorizzato (*Admin*, *Operatore* o *Visualizzatore*). Quando viene aperto un *ticket*, il sistema imposta automaticamente la stazione nello stato “InManutenzione” e invia una notifica *e-mail* (“NuovaManutenzione”) a tutto il personale interessato, assicurando una comunicazione immediata.

2. **Presa in Carico e Pianificazione:** l'intervento appare inizialmente nel diario come "Da Pianificare". Un operatore può prenderlo in carico definendo una data: il sistema registra l'assegnazione e invia la notifica "InterventoPresoInCarico". È prevista anche una gestione collaborativa: altri operatori possono aggiungersi come partecipanti, generando notifiche specifiche ("NuovoPartecipante") per mantenere allineato tutto il *team*.
3. **Chiusura e Esito:** al termine delle attività, l'assegnatario procede alla chiusura dell'intervento, che può avere due esiti distinti:
 - *Risolto con Successo:* la stazione viene rimossa dallo stato di manutenzione, l'intervento viene marcato come chiuso (*IsClosed = true*) e viene inviata una notifica finale con il resoconto delle operazioni.
 - *Non Risolto:* se il problema persiste, l'utente inserisce una nota di fallimento. Il sistema chiude l'intervento corrente ma genera automaticamente un nuovo intervento "figlio". Questo meccanismo mantiene la stazione in manutenzione e preserva l'intera catena storica degli eventi, fondamentale per tracciare guasti complessi o ricorrenti.

L'intero sistema di comunicazione è orchestrato dall'*EmailService*, che recupera i *template HTML* dal *database* e sostituisce dinamicamente i segnaposto (es. {NomeStazione}, {Assegnatario}), garantendo messaggi sempre coerenti e aggiornati.

4. Dettagli implementativi

Una delle decisioni architetturali più rilevanti è stata la progettazione di un'applicazione *event-driven*, in grado di reagire autonomamente agli eventi in tempo reale, senza dipendere dall'interazione dell'utente o dal ciclo di vita delle richieste *HTTP*. Per raggiungere questo obiettivo, il sistema si basa sull'interfaccia *IHostedService* di *ASP.NET Core*, che consente l'esecuzione continua di logiche di *business* critiche in processi di *background* completamente indipendenti dalle operazioni via *web*.

Questi servizi, costantemente operativi, rappresentano il nucleo funzionale dell'automazione, monitorano in modo costante la *share* di rete, effettuano il *parsing* e la normalizzazione dei *file CSV* e rilevano in maniera immediata eventuali condizioni anomale. L'adozione di un'architettura basata su servizi in *background* ha minimizzato la latenza nel rilevamento dei guasti, garantendo una maggiore continuità operativa del servizio. Un approccio basato su un sito *web* "passivo", limitato alla sola logica reattiva delle richieste utente, non sarebbe stato in grado di garantire la stessa *responsiveness*.

4.1 Configurazione dei servizi in *background* (*IHostedService*)

L'architettura del sistema si fonda sull'utilizzo di *IHostedService*, il componente *standard* di *ASP.NET Core*, progettato per eseguire logiche di *business* in *background*. A differenza dei controller *web* tradizionali, questi servizi si avviano contestualmente all'applicazione e rimangono costantemente attivi, operando in modo totalmente indipendente dal traffico *HTTP* in ingresso. Questa caratteristica garantisce un monitoraggio costante e reattivo dell'infrastruttura.

Nel file *Program.cs* sono stati registrati quattro servizi distinti, ciascuno con un perimetro di responsabilità specifico per assicurare la separazione delle logiche e la robustezza del sistema:

CsvProcessingService

- Implementa un ciclo di *polling* temporizzato (1 minuto) basato su *Task.Delay*. La logica critica include l'uso della classe *Impersonation* per l'accesso sicuro alla rete (gestione credenziali *Windows*) e un algoritmo di *parsing* che gestisce automaticamente il fuso

orario UTC e la rotazione delle cartelle giornaliere dei dati sorgente. Include *routine* di *data cleaning* per scartare *record* corrotti senza interrompere il flusso (vedi Appendice B).

OfflineDetectionService

- Utilizza una logica a “doppia chiave” per determinare lo stato. Interroga sia il *database* (*timestamp* “UltimaModifica”) che il *file system* remoto (presenza sismogrammi) per evitare falsi positivi. È l’unico servizio autorizzato a modificare lo stato in “Offline” o “Intermittente”, risolvendo potenziali *race conditions* con il *loader* dei dati (vedi Appendice B).

AlertService

- Implementa una *query LINQ*¹⁸ ottimizzata che filtra le stazioni con tensione sotto soglia (letta dinamicamente da *GlobalSettings*) escludendo quelle già marcate come “Offline”, per evitare notifiche ridondanti su stazioni spente.

HistoricalBackfillService

- Esegue una *complex query* che incrocia i dati di “TensioniStorico” e “Interventi”. Per ottimizzare le performance, utilizza un *flag* nel *database* (*HistoricalBackfillCompleted*) che inibisce l’esecuzione del servizio ai riavvii successivi, risparmiando risorse computazionali.

4.2 Logica di autenticazione ibrida

Uno dei requisiti chiave del sistema era supportare due tipologie di utenti con esigenze distinte: il personale operativo, autenticato tramite credenziali di dominio istituzionale (*Active Directory*), e l’amministratore locale (*account “admin”*), indipendente dall’infrastruttura di rete e utile per configurazioni iniziali o scenari di emergenza. Per soddisfare questa esigenza, la gestione dell’accesso è stata centralizzata nel *controller* di *backend* in *Login.cshtml.cs*, che implementa una logica condizionale rigorosa per garantire la sicurezza prima ancora dell’autenticazione. Il flusso operativo si articola in tre fasi sequenziali come illustrato in Figura 16.

Il flusso di sicurezza segue una procedura rigorosa.

1. **Autorizzazione Preliminare:** il sistema interroga la tabella *AspNetUsers* del *database*. Se lo *username* (es. *ct\utente*) non è censito o risulta sospeso (*IsSuspended = true*), il processo viene interrotto immediatamente, prevenendo attacchi di *brute-force* su *Active Directory*.
2. **Biforcazione dell’Autenticazione:** una volta validata l’identità nel *database*, la logica si divide in due percorsi paralleli:
 - **Utente Locale (Admin):** viene invocato il metodo *PasswordSignInAsync* del *framework ASP.NET Core Identity*, che esegue un confronto crittografico sicuro (*hashing*) con la *password* memorizzata localmente.
 - **Utente di Dominio (Active Directory):** il sistema istanzia un *PrincipalContext* utilizzando la libreria *System.DirectoryServices.AccountManagement*, tramite il metodo *ValidateCredentials()*, le credenziali vengono inviate in tempo reale al *controller di dominio* per la verifica, evitando qualsiasi memorizzazione della *password* nel *database* locale o nei file di *log* per garantire la massima sicurezza.
3. **Creazione della Sessione:** in caso di esito positivo da uno dei due *provider*, il metodo *SignInAsync* genera il *cookie* di autenticazione crittografato, abilitando l’accesso alle risorse protette.

Questa architettura ibrida offre il perfetto equilibrio tra robustezza (autenticazione centralizzata istituzionale) e flessibilità (accesso di emergenza locale), semplificando l’esperienza utente con un’unica interfaccia di accesso trasparente.

¹⁸ LINQ sta per *Language INtegrated Query* (Interrogazione Integrata nel Linguaggio), ed è un potente componente di *Microsoft .NET* che permette di scrivere codice per interrogare dati nel *database*, direttamente all’interno di *C#*.

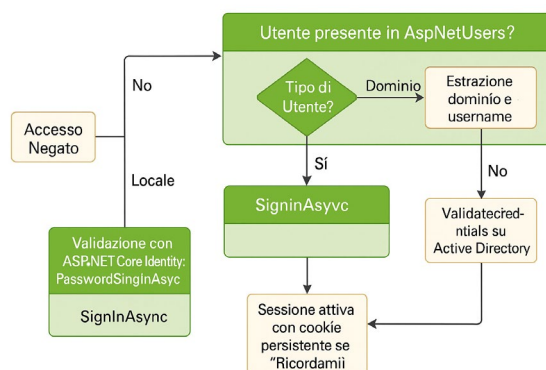


Figura 16 Diagramma di sequenza del Login. Dettaglio della logica decisionale implementata nel controller di autenticazione, che indirizza la richiesta verso il provider corretto (Identity o AD) in base alla tipologia di utente.
Figure 16 Login sequence diagram. Details of the decision-making logic implemented in the authentication controller, which directs the request to the correct provider (Identity or AD) based on the user type.

4.3 Gestione della sicurezza e principio del minimo privilegio

Per garantire un accesso sicuro alle risorse di rete esterne (come la *share \\nasacqmseed*), è stata adottata una strategia rigorosamente allineata al “Principio del minimo privilegio”¹⁹. Elevare i permessi dell'intero *Application Pool* avrebbe introdotto un livello di rischio inaccettabile, poiché una singola vulnerabilità avrebbe potuto compromettere l'intero ambiente applicativo. Per questo motivo, è stato implementato un meccanismo di impersonificazione “Just-in-Time”²⁰. L'applicazione opera normalmente con un *account* a bassissimo privilegio (*ApplicationPoolIdentity*), privo di accesso alla rete, riducendo al massimo la superficie d'attacco. Solo nel momento esatto in cui è necessario accedere a una risorsa remota (es. *CsvProcessingService* o *ImageStreamer*), il servizio esegue una procedura controllata in quattro fasi.

1. **Caricamento credenziali sicure.** Le credenziali di un utente di servizio dedicato vengono lette da *appsettings.json* (vedi Appendice A), posizionato fuori dalla *directory* pubblica del sito per massima sicurezza.
2. **Attivazione impersonificazione.** Una classe *helper* (*Impersonation.cs*) utilizza le API di Windows per far “indossare” al *thread* corrente i permessi dell'utente di dominio.
3. **Esecuzione operazione I/O.** Il servizio compie l'azione richiesta (es. *Directory.Exists* o *File.ReadAllBytes*) nel contesto sicuro dell'utente impersonato.
4. **Rimozione impersonificazione.** Immediatamente dopo l'operazione, il *thread* rilascia i privilegi acquisiti e torna al contesto limitato del *pool*.

Questo modello riduce la finestra di esposizione a pochi millisecondi, poiché per il 99.9% del tempo operativo l'applicazione rimane “blindata” nel suo contesto di sicurezza locale, interagendo con le risorse sensibili di rete solo per l'istante strettamente necessario all'operazione.

¹⁹ *Best practice* di sicurezza che prevede di assegnare agli utenti e ai processi solo i permessi strettamente necessari per svolgere le operazioni richieste.

²⁰ Tecnica di sicurezza che consente a un *thread* di assumere temporaneamente i privilegi di un utente di dominio per accedere a risorse protette, riducendo il rischio di compromissione.

4.4 Data ingestion: gestione robusta dei dati CSV (*data cleaning*)

Il servizio *CsvProcessingService* non si limita a importare i dati: esegue una vera e propria pulizia (*data cleansing*) in tempo reale per garantir il corretto funzionamento del sistema.

I file CSV provenienti dai sistemi di acquisizione possono contenere anomalie, come valori non numerici (es. "N/A" in campi di tensione) o righe corrotte.

Per affrontare queste criticità, abbiamo configurato la libreria *CsvHelper* 21[3] con una logica di gestione degli errori personalizzata.

1. Errori di conversione (*TypeConverterException*)

Se una riga contiene un dato non valido (es. "N/A" nel campo "TensioneBatteria"), l'eccezione viene intercettata e loggata. Solo quella riga viene scartata, consentendo al processo di continuare senza bloccare l'intero file.

2. Dati non validi (*BadDataFound*)

Se una riga presenta un numero errato di colonne, viene anch'essa loggata e ignorata, evitando interruzioni nel flusso di importazione.

Questa configurazione permette di isolare e scartare le singole righe corrotte, assicurando che tutti i dati validi vengano acquisiti senza interrompere il processo di importazione.

Il servizio è stato esteso per elaborare anche i campi geospaziali. I valori vengono validati e convertiti in tipo *double* per garantire la corretta persistenza nel *database*. In caso di valori mancanti o non validi, il sistema assegna *null*, preservando la retrocompatibilità con i *record* esistenti e consentendo l'inserimento di nuove stazioni anche senza dati geospaziali completi.

4.5 Visualizzazione immagini "live" (*ImageStreamer*)

Per la visualizzazione dei sismogrammi e degli spettrogrammi in tempo reale, è stata scartata la soluzione tradizionale di copiare fisicamente i file sul *server web*, poiché avrebbe comportato una massiccia duplicazione dei dati e un utilizzo inefficiente dello spazio disco. La soluzione implementata prevede invece l'uso di *ImageStreamer*, un *controller* dedicato in *ASP.NET Core* che agisce come un *proxy* sicuro tra il *browser* dell'utente e la *share* di rete protetta.

Il flusso operativo è strutturato come segue (vedi Figura 17):

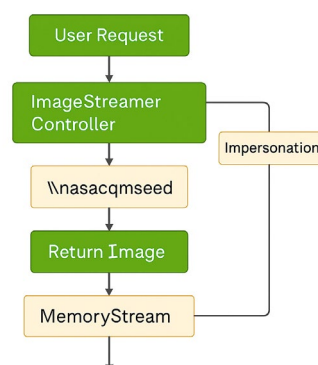
1. *Richiesta Client*: dalla pagina di dettaglio, il *browser* invia una richiesta asincrona all'*endpoint* dedicato (es. */ImageStreamer?filePath=...*), specificando la risorsa desiderata.
2. *Intercettazione e Sicurezza*: il *controller* riceve la chiamata e attiva la procedura di accesso sicuro. Sfruttando la logica di impersonificazione "*Just-in-Time*" (descritta nella Sezione 4.3), il processo acquisisce temporaneamente i permessi di dominio necessari per leggere dalla *share* *\\nasacqmseed*, che altrimenti sarebbe inaccessibile all'applicazione *web*.
3. *Streaming in Memoria*: il file (sia esso un dato *live* appena generato o uno storico) viene letto direttamente dalla sorgente e caricato in un *MemoryStream* volatile.
4. *Trasmissione*: il flusso di *byte* viene trasmesso immediatamente al *browser*, senza mai scrivere il file su disco locale del *server web*.

Questo approccio garantisce che l'operatore visualizzi sempre l'immagine più recente disponibile, eliminando latenze di sincronizzazione e riducendo drasticamente la superficie di attacco, poiché i file originali non lasciano mai la rete interna sicura.

²¹ Libreria .NET per la lettura e scrittura di file CSV con gestione robusta degli errori.

Figura 17 Pipeline di image streaming. Flusso di sicurezza per la visualizzazione delle immagini: il controller *ImageStreamer* utilizza l'impersonificazione temporanea per accedere alla share protetta e trasmettere il file in memoria senza esporre il percorso fisico.

Figure 17 Image streaming pipeline. Security flow for image display: the *ImageStreamer* controller uses temporary impersonation to access the protected share and stream the file to memory without exposing the physical path.



4.6 Logica di definizione dello stato giornaliero

La determinazione univoca dello stato operativo giornaliero per il Calendario di Affidabilità ha rappresentato una delle sfide logiche più complesse del progetto.

Mentre per i dati storici la regola applicata dall'*HistoricalBackfillService* è binaria, la gestione del flusso "Live" richiede un'orchestrazione più raffinata per evitare conflitti tra i servizi concorrenti. Inizialmente l'architettura soffriva di una *race condition* logica e il *CsvProcessingService* impostava lo stato "Online" indiscriminatamente all'arrivo di ogni *file*, sovrascrivendo potenzialmente le rilevazioni di guasto dell'*OfflineDetectionService* e generando incoerenze visive nel *report*.

Per risolvere il problema è stato implementato un protocollo di priorità gerarchica che disciplina come e quando uno stato può essere modificato.

1. **Priorità Assoluta (Manutenzione):** se per una stazione è aperto un *ticket* di intervento, lo stato giornaliero viene forzato su "In Manutenzione" (Giallo), ignorando qualsiasi flusso di dati in ingresso. Questo impedisce che i test tecnici vengano confusi con l'operatività normale.
2. **Inserimento Conservativo (CsvProcessingService):** il servizio di ingestione dati adotta un approccio non distruttivo. Inserisce lo stato "Online" (Verde) solo se non esiste ancora alcun *record* per la giornata corrente. Non ha i permessi per sovrascrivere stati esistenti.
3. **Autorità di Sovrascrittura (OfflineDetectionService):** questo servizio funge da arbitro finale ed è l'unico autorizzato a modificare uno stato preesistente, applicando una logica granulare in caso di interruzione delle trasmissioni:
 - **Offline (Rosso):** se la stazione non ha inviato alcun dato dall'inizio della giornata.
 - **Intermittente (Blu):** se la stazione era precedentemente "Online" (ha inviato dati parziali) ma ha smesso di trasmettere. Questa distinzione è fondamentale per differenziare un *blackout* totale da un problema di connettività.

Questa logica centralizzata garantisce la coerenza tra il monitoraggio in tempo reale e lo storico, offrendo agli operatori una rappresentazione veritiera dell'affidabilità della rete.

4.7 Configurazione dell'ambiente di produzione (IIS)

Per garantire il funzionamento continuo dei servizi in *background* (*IHostedService*), l'*Application Pool* di IIS deve essere configurato in modo diverso rispetto alle impostazioni *standard*, che tendono a sospendere i siti inattivi per risparmiare risorse.

Modifiche per modalità Always-On

- **Timeout di inattività:** impostato a 0 (*default*: 20 min) per evitare lo spegnimento del processo in assenza di traffico.

- **Intervallo di riciclo automatico:** impostato a 0 (default: 1740 min) per disabilitare il riavvio programmato del *pool*.
- **Modalità di avvio:** impostata su *AlwaysRunning* (default: *OnDemand*) per avviare automaticamente l'applicazione e i servizi di monitoraggio al riavvio del *server*, senza attendere la prima richiesta utente.

Gestione Sicurezza Sessioni

Per garantire che un utente *sospeso* venga disconnesso immediatamente:

- gli *handler* *OnPostSuspendUserAsync* e *OnPostDeleteUserAsync* invocano
- *_userManager.UpdateSecurityStampAsync(user)*, invalidando tutti i *cookie* di sessione attivi;
- nel *Program.cs*, l'opzione *SecurityStampValidatorOptions.ValidationInterval* è impostata a *TimeSpan.Zero*.

Di default, *ASP.NET Core* verifica il *Security Stamp* ogni 30 minuti; questa modifica forza il controllo a ogni richiesta, assicurando l'espulsione immediata.

4.8 Generazione *Report PDF* dinamici

Per garantire la producibilità di documentazione ufficiale e cristallizzata dello stato di una stazione in un momento specifico, è stato integrato un modulo di reportistica avanzato. Questa soluzione abbandona l'uso di *template HTML/CSS* statici in favore della libreria *QuestPDF* [5], il documento viene definito interamente tramite codice *C#*, sfruttando un motore di *layout* fortemente tipizzato per comporre il *PDF* in modo dinamico e performante.

Il flusso operativo ("*Pipeline* di Generazione") segue questi step:

1. **Trigger On-Demand:** l'operazione viene invocata dall'operatore direttamente dalla *dashboard*, avviando una richiesta specifica al *backend* per la generazione del *report* istantaneo.
2. **Aggregazione dati Real-Time:** al momento della richiesta, il *controller* istanzia un contesto di esecuzione effimero che aggrega dati eterogenei da fonti diverse:
 - **Telemetria:** recupera l'anagrafica e i parametri di stato più recenti del *database SQL*;
 - **Asset multimediali:** accede alla *share* di rete protetta sfruttando il meccanismo di impersonificazione sicura (vedi Sezione 4.3) per prelevare i dati binari del sismogramma e dello spettro di frequenza attuali;
 - **Audit Log:** estrae l'elenco cronologico completo degli interventi dalla tabella "Diario Interventi".
3. **Rendering e composizione:** i dati strutturati vengono inviati nella classe *StationReportDocument*, che implementa l'interfaccia di generazione. Il motore compone il *layout* vettoriale includendo intestazioni dinamiche, griglie di stato, *rendering* delle immagini e tabelle storiche.
4. **Streaming Stateless:** il documento *PDF* viene generato interamente in *RAM* (su un *MemoryStream*) e trasmesso al *client* come *stream* binario, quindi salvato tramite *browser*.

4.9 Gestione dei dati *time-series* e stato giornaliero

La gestione efficiente di grandi volumi di dati *time-series* rappresenta una sfida critica per le *performance* del sistema: nel nostro caso specifico, la tabella "TensioniStorico" accumula migliaia di nuovi *record* ogni singolo giorno. Per evitare che la generazione del "Calendario di Affidabilità"

richiedesse costose *query* di aggregazione a *runtime*, è stata implementata un'architettura ibrida basata sul *pattern* della "Vista Materializzata" (come illustrato in Figura 18).

La strategia adottata sposta il carico computazionale dalla fase di lettura a quella di scrittura (*Write-Time*²² vs *Read-Time*²³).

1. **Pre-Calcolo (*Write-Time*):** invece di calcolare lo stato della stazione aggregando migliaia di righe nel momento esatto in cui l'utente visita la pagina (*Read-Time*), il calcolo viene delegato ai servizi in *background*.
2. **Persistenza (*StatiGiornalieri*):** il risultato di questo calcolo viene "cristallizzato" in una tabella dedicata (*StatiGiornalieri*), che funge da *cache* persistente dello stato della rete.
3. **Logica Collaborativa:** i servizi operano in cooperazione su questa tabella. Il *CsvProcessingService* imposta lo stato iniziale su "Online" alla ricezione dei dati, mentre l'*OfflineDetectionService* interviene successivamente per sovrascriverlo in "Intermittente" o "Offline" in caso di interruzioni.

Grazie a questo *design*, il caricamento della pagina "RiepilogoStati" non richiede calcoli complessi, ma si riduce a un semplice ed efficiente *SELECT* sulla tabella pre-calcolata. Questo garantisce un *rendering* istantaneo dell'interfaccia anche in presenza di milioni di *record* storici, riducendo drasticamente il carico sulla *CPU* del *database server*.

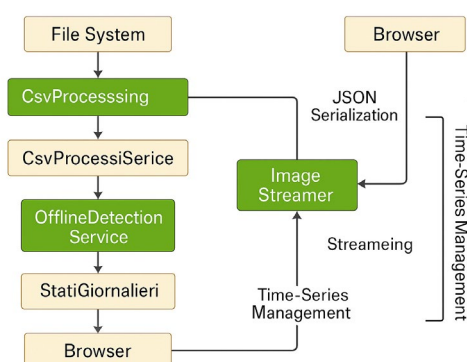


Figura 18 Gestione dati *Time-Series*. Flusso completo del dato, dall'acquisizione su *File System* alla visualizzazione, evidenziando la strategia di pre-calcolo "*Write-Time*" per la tabella "*StatiGiornalieri*".

Figure 18 *Time-Series data management*. Complete data flow, from file system acquisition to visualization, highlighting the "*Write-Time*" pre-calculation strategy for the "*StatiGiornalieri*" table.

5. Validazione e risoluzione delle criticità

La validazione della piattaforma è stata condotta seguendo un approccio sistematico e orientato alla robustezza, con l'obiettivo di assicurare la stabilità operativa anche in condizioni di carico elevato o in presenza di dati incompleti. Oltre ai *test* funzionali *standard*, sono state eseguite prove di *stress* per verificare la stabilità dei servizi in *background* sotto carico.

²² *Write-Time* pre-calcolo e memorizzazione dei dati al momento della scrittura per ridurre il carico in fase di lettura.

²³ *Read-Time* calcolo dei dati al momento della richiesta, più flessibile ma meno performante.

5.1 Validazione dei servizi in *background* (*fault tolerance*)

Il cuore del sistema, costituito dai servizi *IHostedService*, è stato validato simulando scenari di guasto comuni nell'infrastruttura di rete.

Gestione della concorrenza sui file. Durante i test di integrazione è stato simulato l'accesso simultaneo ai file mentre questi erano ancora in fase di scrittura da parte dei sistemi di acquisizione. Per evitare eccezioni bloccanti (*IOException*) è stata implementata nel *CsvProcessingService* una logica di filtraggio che ignora i file temporanei (estensione *.tmp*) e gestisce i tentativi di lettura con un meccanismo di *retry* al ciclo successivo. Questo accorgimento previene il *crash* del servizio in caso di file bloccati dal sistema operativo.

Robustezza del Parsing (*data cleansing*). L'esecuzione di test su *dataset* reali ha messo in evidenza la frequente presenza di valori anomali nei file CSV (come campi "N/A", formati incoerenti o righe corrotte). Per evitare che tali anomalie compromettessero l'intero processo, è stato adottato un modello di *parsing* resiliente. Il *parser* intercetta in modo puntuale le eccezioni, isola i *record* non validi e procede comunque con l'elaborazione dei dati rimanenti. In questo modo il sistema mantiene la continuità del monitoraggio anche in presenza di *input* parzialmente danneggiati, riducendo drasticamente il rischio di interruzioni dovute a dati da considerare non validi.

5.2 Validazione della logica di stato (*race conditions*)²⁴

Durante le prove è emersa una criticità legata al sincronismo tra il servizio di importazione dei dati e il modulo di rilevamento degli stati *offline*. L'esecuzione concorrente delle due *routine* su *thread* paralleli generava conflitti di scrittura sulla tabella degli stati.

Nella prima implementazione, entrambi i servizi tentavano di aggiornare il campo "Stato" contemporaneamente. Poteva verificarsi una situazione in cui il "*CsvProcessingService*" impostava lo stato su "*Online*" nell'esatto istante in cui l'"*OfflineDetectionService*" stava scrivendo "*Offline*". L'esito finale dipendeva dall'ordine casuale di *commit* delle transazioni sul *database*, causando un fenomeno di *flapping*, lo stato della stazione oscillava tra "*Online*" e "*Offline*" in pochi millisecondi.

Per risolvere il problema in modo deterministico, la logica di scrittura è stata riprogettata separando le responsabilità, il "*CsvProcessingService*" si limita ad aggiornare esclusivamente il *timestamp* dell'ultima ricezione, senza mai modificare direttamente lo stato operativo. L'"*OfflineDetectionService*" è l'unico che può modificare il campo "Stato". Ad ogni ciclo, analizza i *timestamp* aggiornati dagli altri servizi e calcola lo stato finale. Questa modifica ha eliminato alla radice la possibilità di conflitto, garantendo che lo stato visualizzato sia sempre coerente con l'ultimo dato temporale disponibile.

5.3 Gestione delle criticità in produzione: caso di studio (*Schema Drift*)

Il *deployment* in ambiente di produzione (*IIS*) ha rappresentato un importante banco di prova per le procedure di rilascio. Durante il primo avvio, il sistema ha riscontrato un errore critico (*HTTP Error 500.30*) causato da uno *Schema Drift*²⁵ (disallineamento tra codice e *database*) (come illustrato in Figura 19). L'analisi dei *log* ha evidenziato un'eccezione SQL relativa a colonne

²⁴ Condizione di anomalia *software* in cui il risultato di un processo dipende dall'ordine non controllabile di eventi concorrenti.

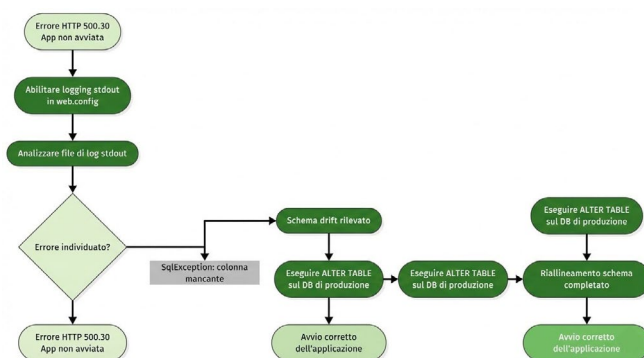
²⁵ Disallineamento tra la struttura del *database* in produzione (tabelle, colonne, tipi di dato) e il modello dati atteso dal codice applicativo.

mancanti (*SQLException*). Questo episodio ha permesso di validare e formalizzare la procedura di risoluzione delle emergenze.

- **Diagnosi:** attivazione dei *log* dettagliati in *web.config* per isolare l'errore.
- **Risoluzione:** esecuzione di *script* di migrazione manuale sul *database* di produzione.
- **Prevenzione:** è stata introdotta una procedura di *scripting* delle migrazioni obbligatoria prima di ogni aggiornamento, garantendo che lo schema del *database* sia sempre sincronizzato prima del riavvio dell'applicazione.

Figura 19 Workflow di *troubleshooting*. Diagramma decisionale utilizzato per la diagnosi e risoluzione dell'errore HTTP 500.30 (Schema Drift) durante il *deploy* in produzione.

Figure 19 Troubleshooting workflow. Decision tree used to diagnose and resolve HTTP Error 500.30 (Drift Schema) during production deployment.



6. Conclusioni

L'applicazione ha centralizzato le operazioni di monitoraggio dello stato di funzionamento della Rete Sismica Mobile dell'OE, e di analisi di base dei dati, sostituendo le procedure manuali con un sistema di controllo automatizzato e notifiche in tempo reale. L'architettura *backend*, fondata su servizi in *background* (*IHostedService*), garantisce ora un'elaborazione dei dati ininterrotta e una rilevazione pressoché istantanea delle anomalie, superando i limiti di latenza dei sistemi precedenti. Sotto il profilo delle *performance*, l'implementazione di una logica di **storizzazione degli stati giornalieri** (calcolo *write-time*) si è rivelata decisiva. Questa strategia evita di dover rielaborare l'intero storico a ogni richiesta dell'utente, garantendo così tempi di risposta immediati e un'interfaccia sempre reattiva, indipendentemente dalla quantità di dati accumulati nel tempo. Sul fronte dell'esperienza utente, l'integrazione della mappa interattiva nella *Dashboard* offre un valore aggiunto fondamentale. La visualizzazione geospaziale, grazie ai *marker* che cambiano colore in tempo reale, permette agli operatori di monitorare l'intera rete con un solo colpo d'occhio, individuando immediatamente le stazioni critiche senza dover analizzare lunghe liste di dati.

In sintesi, la piattaforma non si limita a ottimizzare i tempi di reazione ai guasti, ma fornisce strumenti avanzati per la manutenzione, elevando sensibilmente l'operativa e l'affidabilità complessiva del servizio di monitoraggio.

7. Disponibilità del codice sorgente

In risposta alle esigenze di trasparenza, replicabilità e riutilizzo da parte di altri gruppi di ricerca dell'istituto, l'intero codice sorgente del progetto è stato pubblicato sul *repository* "<https://gitlab.rm.ingv.it/carmelo.lapiana/gestionestazionisismiche>" con licenza *Open Source*. Le Appendici di questo documento riportano, per comodità di consultazione, solo gli estratti più significativi del codice, rimandando al *repository* per la versione completa e mantenuta.

Contributi degli autori

Come autore principale C.L.P. ha portato avanti l'intero sviluppo del *software* presentato, curandone le scelte implementative, che sono andate dalla selezione dei *framework* da utilizzare, fino ai *pattern* architetturali. M.T. ha contribuito alla revisione macroscopica dell'evoluzione del progetto, interagendo, in alcune fasi, con lo sviluppatore. F.P. ha dato il suo contributo per la redazione del presente rapporto tecnico.

Specifiche tecniche e risorse software

Per l'implementazione della piattaforma sono state utilizzate diverse librerie e *framework* consolidati, tra cui:

- **Microsoft (2024).** *ASP.NET Core 8 Documentation*. Guida ufficiale allo sviluppo di applicazioni web con .NET 8 e gestione della sicurezza (Identity).
- **Microsoft (2024).** *System.DirectoryServices.AccountManagement Namespace*. Documentazione delle API per l'integrazione con Active Directory Domain Services.
- **Josh Close (2024).** *CsvHelper*. Libreria .NET per la lettura e scrittura rapida e robusta di file CSV.
- **ApexCharts.** *ApexCharts.js*. Libreria JavaScript open-source per la creazione di grafici interattivi e responsive.
- **QuestPDF.** *QuestPDF Library*. Libreria .NET open-source per la generazione di documenti PDF tramite layout engine fluido.
- **Volodymyr Agafonkin.** *Leaflet*. Libreria JavaScript leggera per mappe interattive mobile-friendly.
- **Bootstrap Team.** *Bootstrap 5*. Framework frontend per lo sviluppo di interfacce responsive e mobile-first.
- **FancyApps.** *Fancybox*. Libreria JavaScript per la visualizzazione di immagini e contenuti multimediali in stile "lightbox".

Sitografia

- [1] Microsoft ASP.NET Core: <https://learn.microsoft.com/aspnet/core>
- [2] System.DirectoryServices.AccountManagement: <https://learn.microsoft.com/dotnet/api/system.directoryservices.accountmanagement>
- [3] CsvHelper Documentation: <https://joshclose.github.io/CsvHelper/>
- [4] ApexCharts Documentation: <https://apexcharts.com/docs/>
- [5] QuestPDF Documentation: <https://www.questpdf.com/>
- [6] Leaflet.js - Interactive Maps: <https://leafletjs.com/>
- [7] Bootstrap 5 Documentation: <https://getbootstrap.com/docs/5.0/getting-started/introduction/>
- [8] Fancybox Documentation: <https://fancyapps.com/fancybox/>

APPENDICI

Appendice A

Mostra come configurare il sistema usando il file *appsettings.json*.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost\\SQLEXPRESS;Database=*****;Trusted_
Connection=True;TrustServerCertificate=True"
  },
  "NetworkShare": {
    "Username": "*****",
    "Password": "*****",
    "Domain": "ct.int.ingv.it"
  },
  "Admins": [
    {
      "Username": "ct\\*****",
      "Email": "*****@ingv.it"
    }
  ],
  "EmailSettings": {
    "SmtpServer": "smtp.gmail.com",
    "SmtpPort": 587,
    "SmtpUser": "*****@gmail.com",
    "SmtpPass": "*****",
    "FromAddress": "*****@gmail.com",
    "AdminEmailForAlerts": "*****@ingv.it"
  },
  "AllowedHosts": "*"
}
```

Appendice B

Questo servizio si occupa dell'acquisizione, pulizia e inserimento dei dati CSV. È responsabile dell'aggiornamento dello stato in tempo reale e della registrazione dello stato 'Online' nel riepilogo giornaliero.

Ecco il codice di *CsvProcessingService.cs*:

```
using CsvHelper;
using CsvHelper.Configuration;
using CsvHelper.TypeConversion;
using GestioneStazioniSismiche.Data;
```

```

using GestioneStazioniSismiche.Models;
using GestioneStazioniSismiche.Security;
using GestioneStazioniSismiche.Utills;
using Microsoft.EntityFrameworkCore;
using System.Globalization;
using System.Runtime.Versioning;

namespace GestioneStazioniSismiche.Services
{

    public class CsvRecordMap : ClassMap<CsvRecord>
    {
        public CsvRecordMap()
        {
            Map(m => m.Timestamp).Index(0);
            Map(m => m.Rete).Index(1);
            Map(m => m.NomeStazione).Index(2);
            Map(m => m.TensioneBatteria).Index(3).TypeConverter<NullableDoubleConverter>();
            Map(m => m.Temperatura).Index(4).TypeConverter<NullableDoubleConverter>();
            Map(m => m.Umidita).Index(5).TypeConverter<NullableDoubleConverter>();
            Map(m => m.StatoGps).Index(6);
            Map(m => m.NumeroSatellitiLock).Index(7).TypeConverter<NullableIntConverter>();
            Map(m => m.NumeroSatellitiVisibili).Index(8).TypeConverter<NullableIntConverter>();
            Map(m => m.Latitudine).Index(9).TypeConverter<NullableDoubleConverter>();
            Map(m => m.Longitudine).Index(10).TypeConverter<NullableDoubleConverter>();
            Map(m => m.Altitudine).Index(11).TypeConverter<NullableDoubleConverter>();
        }
    }

    public class CsvProcessingService : BackgroundService
    {
        private readonly IServiceProvider _serviceProvider;
        private readonly ILogger<CsvProcessingService> _logger;
        private readonly IConfiguration _configuration;
        private const string CsvSourcePath = @"\\nasacqmseed";

        public CsvProcessingService(IServiceProvider serviceProvider,
            ILogger<CsvProcessingService> logger, IConfiguration configuration)
        {
            _serviceProvider = serviceProvider;
            _logger = logger;
            _configuration = configuration;
        }

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation("Servizio CSV avviato. Monitoraggio di: {Path}",
                CsvSourcePath);
            await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);
            while (!stoppingToken.IsCancellationRequested)

```

```

    {
        try { await CheckForNewCsvData(stoppingToken); }
        catch (Exception ex) { _logger.LogError(ex, "Errore critico non gestito nel ciclo del servizio CSV."); }
        await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
    }
}

private DateTime ConvertFromUnixTimestamp(double timestamp) => DateTime.
UnixEpoch.AddSeconds(timestamp);

private async Task CheckForNewCsvData(Cancellation_token stoppingToken)
{
    using var scope = _serviceProvider.CreateScope();
    var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

    var today = DateTime.UtcNow;
    var dataGiorno = DateOnly.FromDateTime(today);
    var dailyFolderName = today.ToString("yyyyMMdd");
    var dailyFolderPath = Path.Combine(CsvSourcePath, today.ToString("yyyy"), "RSM_
products", "csvfiles", dailyFolderName);

    var fileContents = new Dictionary<string, string>();
    try
    {
        var username = _configuration["NetworkShare:Username"];
        var password = _configuration["NetworkShare:Password"];
        var domain = _configuration["NetworkShare:Domain"];

        var impersonation = new Impersonation(username, domain, password);
        impersonation.Run(() =>
        {
            if (!Directory.Exists(dailyFolderPath))
            {
                var yesterday = today.AddDays(-1);
                var yesterdayFolderName = yesterday.ToString("yyyyMMdd");
                var yesterdayFolderPath = Path.Combine(CsvSourcePath, yesterday.
ToString("yyyy"), "RSM_products", "csvfiles", yesterdayFolderName);

                if (Directory.Exists(yesterdayFolderPath))
                {
                    ReadFilesFromPath(yesterdayFolderPath, ref fileContents);
                }
            }
            else
            {
                ReadFilesFromPath(dailyFolderPath, ref fileContents);
            }
        });
    }
}

```

```

        catch (Exception ex) { _logger.LogError(ex, "Accesso alla rete fallito. Percorso: {path}",
dailyFolderPath); return; }

        if (!fileContents.Any()) return;

        var stazioniDict = await dbContext.StazioniSismiche.ToDictionaryAsync(s =>
s.NomeStazione, s => s, StringComparer.OrdinalIgnoreCase, stoppingToken);
        var retiSet = new HashSet<string>(await dbContext.Reti.Select(r => r.NomeRete).
ToListAsync(stoppingToken), StringComparer.OrdinalIgnoreCase);

        foreach (var fileEntry in fileContents)
        {
            var fileName = Path.GetFileName(fileEntry.Key);
            var nomeStazioneDalFile = fileName.Split('_').Length > 2 ? fileName.Split('_')[2] : null;
            if (string.IsNullOrEmpty(nomeStazioneDalFile)) continue;

            var recordsInFile = new List<CsvRecord>();
            try
            {
                var config = new CsvConfiguration(CultureInfo.InvariantCulture) {
HasHeaderRecord = false, Delimiter = ",", BadDataFound = context => { _logger.
LogWarning("Riga CSV non valida nel file {FileName}. Riga: {RawRecord}", fileName, context.
RawRecord); } };
                using var reader = new StringReader(fileEntry.Value);
                using var csv = new CsvReader(reader, config);
                csv.Context.RegisterClassMap<CsvRecordMap>();

                while (csv.Read())
                {
                    try
                    {
                        var record = csv.GetRecord<CsvRecord>();
                        if (record != null) recordsInFile.Add(record);
                    }
                    catch (TypeConverterException ex) { _logger.LogWarning(ex, "Errore di
conversione dati nel file {FileName} alla riga {RawRow}: {RawRecord}", fileName, csv.Parser.
RawRow, csv.Parser.RawRecord); }
                }
            }
            catch (Exception ex) { _logger.LogError(ex, "Errore critico durante il parsing del file
CSV {FileName}", fileName); continue; }

            if (!recordsInFile.Any()) continue;

            var validRecords = recordsInFile.Where(r => r.TensioneBatteria.HasValue).ToList();
            if (!validRecords.Any()) continue;

            if (!stazioniDict.TryGetValue(nomeStazioneDalFile, out var stazioneDb))
            {
                var reteDelFile = validRecords.FirstOrDefault()?.Rete;

```

```

        if (!string.IsNullOrEmpty(reteDelFile) && !retiSet.Contains(reteDelFile))
        {
            dbContext.Reti.Add(new Rete { NomeRete = reteDelFile });
            retiSet.Add(reteDelFile);
        }
        stazioneDb = new StazioneSismica { NomeStazione = nomeStazioneDalFile, Rete
= reteDelFile };
        stazioneDb.DataInstallazione = ConvertFromUnixTimestamp(validRecords.
OrderBy(r => r.Timestamp).First().Timestamp);
        dbContext.StazioniSismiche.Add(stazioneDb);
        stazioniDict[nomeStazioneDalFile] = stazioneDb;
    }

    var statoGiornaliero = await dbContext.StatiGiornalieri.FirstOrDefaultAsync(s =>
s.NomeStazione == stazioneDb.NomeStazione && s.Data == dataGiorno, stoppingToken);
    if (statoGiornaliero == null)
    {
        statoGiornaliero = new StatoGiornaliero { NomeStazione = stazioneDb.
NomeStazione, Data = dataGiorno, Stato = "Online" };
        await dbContext.StatiGiornalieri.AddAsync(statoGiornaliero, stoppingToken);
    }
    else if (statoGiornaliero.Stato == "Offline")
    {
        statoGiornaliero.Stato = "Intermittente";
    }
    else if (statoGiornaliero.Stato != "Manutenzione")
    {
        statoGiornaliero.Stato = "Online";
    }

    var ultimoTimestampSalvato = stazioneDb.UltimaModifica;
    var newRecords = validRecords.Where(r => ultimoTimestampSalvato == null ||
ConvertFromUnixTimestamp(r.Timestamp) > ultimoTimestampSalvato).OrderBy(r =>
r.Timestamp).ToList();

    if (!newRecords.Any())
    {
        await dbContext.SaveChangesAsync(stoppingToken);
        continue;
    }

    var lastRecord = newRecords.Last();
    stazioneDb.TensioneBatteria = lastRecord.TensioneBatteria.Value;
    stazioneDb.Temperatura = lastRecord.Temperatura;
    stazioneDb.Umidita = lastRecord.Umidita;
    stazioneDb.StatoGps = lastRecord.StatoGps;
    stazioneDb.NumeroSatellitiLock = lastRecord.NumeroSatellitiLock;
    stazioneDb.NumeroSatellitiVisibili = lastRecord.NumeroSatellitiVisibili;
    stazioneDb.UltimaModifica = ConvertFromUnixTimestamp(lastRecord.Timestamp);

```

```

        if (lastRecord.Latitudine.HasValue)
            stazioneDb.Latitudine = lastRecord.Latitudine;
        if (lastRecord.Longitudine.HasValue)
            stazioneDb.Longitudine = lastRecord.Longitudine;
        if (lastRecord.Altitudine.HasValue)
            stazioneDb.Altitudine = lastRecord.Altitudine;

        var nuoviRecordStorico = newRecords.Select(r => new TensioneStorico
        {
            NomeStazione = stazioneDb.NomeStazione,
            Timestamp = ConvertFromUnixTimestamp(r.Timestamp),
            ValoreTensione = r.TensioneBatteria.Value
        }).ToList();

        await dbContext.TensioniStorico.AddRangeAsync(nuoviRecordStorico,
stoppingToken);
        await dbContext.SaveChangesAsync(stoppingToken);
    }
}

[SupportedOSPlatform("windows")]
private void ReadFilesFromPath(string directoryPath, ref Dictionary<string, string>
fileContents)
{
    var filePaths = Directory.GetFiles(directoryPath, "*.csv");
    foreach (var path in filePaths)
    {
        try
        {
            using var fileStream = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);
            using var streamReader = new StreamReader(fileStream);
            fileContents[path] = streamReader.ReadToEnd();
        }
        catch (IOException ex) { _logger.LogWarning(ex, "Impossibile leggere il file
{FileName} perché è bloccato.", Path.GetFileName(path)); }
    }
}
}

```

Questo servizio agisce come guardiano della rete. Rileva le stazioni che non comunicano, invia gli allarmi e-mail e aggiorna il riepilogo giornaliero impostando gli stati 'Offline' o 'Intermittente' in caso di guasto."

Ecco il codice di *OfflineDetectionService.cs*:

```

using GestioneStazioniSismiche.Data;
using GestioneStazioniSismiche.Models;
using GestioneStazioniSismiche.Security;

```

```

using Microsoft.EntityFrameworkCore;
using System.Globalization;
using Microsoft.AspNetCore.Identity;
using System.Runtime.Versioning;

namespace GestioneStazioniSismiche.Services
{
    public class OfflineDetectionService : BackgroundService
    {
        private readonly IServiceProvider _serviceProvider;
        private readonly ILogger<OfflineDetectionService> _logger;
        private readonly IConfiguration _configuration;
        private const string NetworkSourcePath = @"\\nasacqmseed";

        public OfflineDetectionService(IServiceProvider serviceProvider,
        ILogger<OfflineDetectionService> logger, IConfiguration configuration)
        {
            _serviceProvider = serviceProvider;
            _logger = logger;
            _configuration = configuration;
        }

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation("Servizio di Rilevamento Stazioni Offline avviato.");
            await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken); // Attesa iniziale

            while (!stoppingToken.IsCancellationRequested)
            {
                try
                {
                    await CheckStationsStatus(stoppingToken);
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Errore non gestito nel servizio di rilevamento offline.");
                }

                await Task.Delay(TimeSpan.FromMinutes(15), stoppingToken); // Controllo periodico
            }
        }

        [SupportedOSPlatform("windows")]
        private async Task CheckStationsStatus(CancellationToken stoppingToken)
        {
            _logger.LogInformation("Esecuzione controllo stato stazioni (Logica CSV + Sismogramma)...");

            using var scope = _serviceProvider.CreateScope();
            var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

```

```

var emailService = scope.ServiceProvider.GetRequiredService<EmailService>();
var userManager = scope.ServiceProvider.
GetRequiredService<userManager<ApplicationUser>>();

var today = DateTime.UtcNow;
var dataGiorno = DateOnly.FromDateTime(today);

var offlineSetting = await dbContext.AppSettings.FirstOrDefaultAsync(s => s.Key ==
"Intervallo Stazione Offline (Ore)", stoppingToken);
double offlineHours = double.TryParse(offlineSetting?.Value, CultureInfo.InvariantCulture,
out var hours) ? hours : 1.0;
var offlineThreshold = TimeSpan.FromHours(offlineHours);

var stazioniDaControllare = await dbContext.StazioniSismiche
.Where(s => !s.IsDismessa && !s.InManutenzione)
.ToListAsync(stoppingToken);

// Accesso alla rete per sismogrammi
var username = _configuration["NetworkShare:Username"];
var password = _configuration["NetworkShare:Password"];
var domain = _configuration["NetworkShare:Domain"];

if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password) || string.
IsNullOrEmpty(domain))
{
    _logger.LogError("Credenziali di rete non configurate. Controllo offline userà solo dati
CSV.");
}

// Dizionario per ultimo sismogramma per stazione
var latestSismogrammi = new Dictionary<string, DateTime>();

try
{
    var impersonation = new Impersonation(username, domain, password);
    impersonation.Run(() =>
    {
        var dailyFolderName = today.ToString("yyyyMMdd");
        var dailyFolderPath = Path.Combine(NetworkSourcePath, today.ToString("yyyy"),
"sismogrammi_RSM", dailyFolderName);

        if (!Directory.Exists(dailyFolderPath))
        {
            _logger.LogWarning("Cartella sismogrammi del giorno non ancora creata: {Path}",
dailyFolderPath);
            return;
        }

        var imageFiles = Directory.GetFiles(dailyFolderPath, "*.png");
    }
}

```

```

        foreach (var nomeStazione in stazioniDaControllare.Select(s => s.NomeStazione))
        {
            var latestFileOnShare = imageFiles
                .Where(f => Path.GetFileName(f).Contains(nomeStazione, StringComparison.
OrdinallgnoreCase))
                .Select(f => new FileInfo(f).LastWriteTimeUtc)
                .OrderByDescending(t => t)
                .FirstOrDefault();

            if (latestFileOnShare != default)
                latestSismogrammi[nomeStazione] = latestFileOnShare;
        }
    });
}
catch (Exception ex)
{
    _logger.LogError(ex, "Accesso alla rete fallito. Controllo offline solo su CSV.");
}

foreach (var stazione in stazioniDaControllare)
{
    bool eraOffline = stazione.IsOffline;

    // Tempo dall'ultimo CSV
    var tempoTrascorsoCSV = stazione.UltimaModifica.HasValue ? DateTime.UtcNow
- stazione.UltimaModifica.Value : TimeSpan.MaxValue;

    // Tempo dall'ultimo PNG
    latestSismogrammi.TryGetValue(stazione.NomeStazione, out var ultimoSismogramma);
    var tempoTrascorsoSismogramma = (ultimoSismogramma != default) ? DateTime.UtcNow
- ultimoSismogramma : TimeSpan.MaxValue;

    // Offline solo se ENTRAMBI i segnali mancano
    bool isOffline = (tempoTrascorsoCSV > offlineThreshold) &&
(tempoTrascorsoSismogramma > offlineThreshold);

    if (isOffline)
    {
        // Aggiorna stato stazione e invia email se necessario
        if (!eraOffline)
        {
            _logger.LogWarning("Stazione {Stazione} è OFFLINE. (CSV: {TempoCSV},
Sismogramma: {TempoSismogramma})",
                stazione.NomeStazione, tempoTrascorsoCSV, tempoTrascorsoSismogramma);
            stazione.IsOffline = true;
            dbContext.Update(stazione);
            await InviaEmailDiNotifica(stazione, "StazioneOffline", userManager, emailService,
offlineHours);
        }
    }
}

```

```

    var statoGiornaliero = await dbContext.StatiGiornalieri
        .FirstOrDefaultAsync(s => s.NomeStazione == stazione.NomeStazione && s.Data ==
dataGiorno, stoppingToken);

    if (statoGiornaliero == null)
    {
        statoGiornaliero = new StatoGiornaliero { NomeStazione = stazione.NomeStazione,
Data = dataGiorno, Stato = "Offline" };
        await dbContext.StatiGiornalieri.AddAsync(statoGiornaliero, stoppingToken);
    }
    else if (statoGiornaliero.Stato == "Online")
    {
        // Transizione Online → Offline → Intermittente
        statoGiornaliero.Stato = "Intermittente";
    }
    else if (statoGiornaliero.Stato != "Manutenzione")
    {
        statoGiornaliero.Stato = "Offline";
    }
}
else
{
    // Stazione ONLINE
    if (eraOffline)
    {
        _logger.LogInformation("Stazione {Stazione} è tornata ONLINE.", stazione.
NomeStazione);
        stazione.IsOffline = false;
        dbContext.Update(stazione);
        await InviaEmailDiNotifica(stazione, "StazioneOnline", userManager, emailService);
    }

    // Aggiorna stato giornaliero
    var statoGiornaliero = await dbContext.StatiGiornalieri
        .FirstOrDefaultAsync(s => s.NomeStazione == stazione.NomeStazione && s.Data ==
dataGiorno, stoppingToken);

    if (statoGiornaliero == null)
    {
        statoGiornaliero = new StatoGiornaliero { NomeStazione = stazione.NomeStazione,
Data = dataGiorno, Stato = "Online" };
        await dbContext.StatiGiornalieri.AddAsync(statoGiornaliero, stoppingToken);
    }
    else if (statoGiornaliero.Stato != "Manutenzione")
    {
        statoGiornaliero.Stato = "Online";
    }
}
}
}

```

```

        await dbContext.SaveChangesAsync(stoppingToken);
    }

    private async Task InviaEmailDiNotifica(
        StazioneSismica stazione,
        string templateKey,
        UserManager<ApplicationUser> userManager,
        EmailService emailService,
        double? oreOffline = null)
    {
        _logger.LogInformation("Invio notifica {TemplateKey} per {Stazione}", templateKey, stazione.
NomeStazione);

        // Trova destinatari: Admin + utenti con permesso sulla rete
        var admins = await userManager.GetUsersInRoleAsync("Admin");

        using var scope = _serviceProvider.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

        var userIdsConPermesso = await dbContext.UserRetePermissions
            .Where(p => p.Rete == stazione.Rete)
            .Select(p => p.UserId)
            .ToListAsync();

        var utentiConPermesso = await userManager.Users
            .Where(u => userIdsConPermesso.Contains(u.Id))
            .ToListAsync();

        var destinatari = admins.Union(utentiConPermesso).DistinctBy(u => u.Id);

        var placeholders = new Dictionary<string, string>
        {
            { "{NomeStazione}", stazione.NomeStazione },
            { "{OreOffline}", oreOffline?.ToString("F0") ?? "N/A" },
            { "{UltimoAggiornamento}", stazione.UltimaModifica?.ToLocalTime().ToString("g") ?? "N/D" }
        };

        foreach (var user in destinatari)
        {
            if (!string.IsNullOrEmpty(user.Email) && user.NotifyOnMaintenance)
            {
                await emailService.SendEmailFromTemplateAsync(user.Email, templateKey,
placeholders);
            }
        }
    }
}

```




QUADERNI di GEOFISICA

ISSN 1590-2595

<https://collane-editoriali.ingv.it/index.php/qdg/it/index/>

QUADERNI di GEOFISICA è una rivista scientifica che pubblica dati, misurazioni e osservazioni geofisiche (anche preliminari) in italiano e inglese. L'obiettivo è una diffusione rapida e gratuita per la comunità scientifica, garantita da un processo di peer-review e da un Editorial Board multidisciplinare. La rivista è inclusa in "Emerging Sources Citation Index" di Clarivate Analytics e "Open Access Journals" di Scopus, e adotta il modello Diamond Open Access, che non prevede costi né per gli autori né per i lettori.

QUADERNI di GEOFISICA is a scientific journal that publishes geophysical data, measurements and observations, including preliminary results, in Italian and English. Its aim is to ensure rapid and free dissemination to the research community, supported by a peer-review process and by a multidisciplinary Editorial Board. The journal is included in the Emerging Sources Citation Index of Clarivate Analytics and it's included in the Open Access journals of Scopus. It follows the Diamond Open Access model, with no charges for either authors or readers.



RAPPORTI TECNICI INGV

ISSN 2039-7941

<https://collane-editoriali.ingv.it/index.php/rt/it/index>

RAPPORTI TECNICI INGV è una rivista tecnico-scientifica che mette in mostra le numerose attività tecnologiche e di ricerca condotte dai ricercatori e tecnici INGV, e non solo. Pubblicata in italiano e inglese, RAPPORTI TECNICI INGV è ideata per valorizzare anche quei lavori che offrono un contributo significativo dal punto di vista tecnico, operativo o documentale fondamentale per lo sviluppo dell'Ente. La rivista adotta il modello Diamond Open Access, che non prevede costi né per gli autori né per i lettori.

RAPPORTI TECNICI INGV is a technical-scientific journal that showcases the wide range of technological and research activities carried out by INGV researchers and technical staff, as well as by external contributors. Published in Italian and English, RAPPORTI TECNICI INGV is designed to highlight works that provide a significant contribution from a technical, operational, or documentary perspective, and that are essential to INGV development. The journal follows the Diamond Open Access model, with no charges for either authors or readers.



MISCELLANEA INGV

ISSN 2039-6651

<https://collane-editoriali.ingv.it/index.php/mi/it/index>

MISCELLANEA INGV rappresenta una tipologia di editoria non tradizionalmente catalogabile nelle ordinarie riviste scientifiche di settore. Pubblicata in lingua italiana e inglese, MISCELLANEA INGV offre uno spazio per la pubblicazione di materiali eterogenei che, pur non rientrando nei formati più strutturati delle altre Collane editoriali INGV, rivestono un interesse scientifico, tecnico o divulgativo. La rivista adotta il modello Diamond Open Access, che non prevede costi né per gli autori né per i lettori.

MISCELLANEA INGV represents a form of publishing that cannot be classified within the standard scientific journals. Published in Italian and English, MISCELLANEA INGV provides a venue for the publication of heterogeneous materials that are of scientific, technical or outreach interest, even though they do not fall within the more structured formats of the other INGV editorial series. The journal follows the Diamond Open Access model, with no charges for either authors or readers.

Progetto grafico
Barbara ANGIONI
Istituto Nazionale di Geofisica e Vulcanologia

Ufficio Editoriale
Francesca DI STEFANO | Coordinatore

Segreteria di redazione
segreteria.collane-editoriali@ingv.it

Editing | Produzione | Grafica redazionale
Barbara ANGIONI
Massimiliano CASCONI
Francesca DI STEFANO
Patrizia PANTANI

© 2026 the Author(s). Open Access.
This article is licensed under a Creative Commons Attribution 4.0 International License.



Creative Commons Attribution 4.0 International



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA