MOPA

Applicativo web per il monitoraggio degli strumenti collegati alle attività della Parabola G. Grueff della stazione radioastronomica di Medicina

Autore: Andrea Atza

Referee: Andrea Orlati

Bologna, 2026

INDICE

1. INTRODUZIONE

MOPA è un'applicazione single page di monitoraggio e controllo sviluppata per il Radiotelescopio di Medicina. Il sistema fornisce un'interfaccia web accessibile da browser per l'acquisizione e la visualizzazione in tempo reale dei dati provenienti dalle IF dei ricevitori, attraverso l'uso di due analizzatori di spettro, e una panoramica generale della condizione operativa dell'antenna.

L'interfaccia consente agli operatori, previa autenticazione, di controllare i parametri di acquisizione degli strumenti, confrontare le polarizzazioni ricevute, monitorare lo stato dell'antenna e salvare i dati spettrali con alcuni metadati (UTC, posizione azimutale ed elevazione dell'antenna).

Il presente rapporto descrive l'architettura software, le componenti principali e le procedure operative.

2. ARCHITETTURA

2.1. Panoramica

L'applicativo è strutturato in tre moduli Python. Il framework principale utilizzato è NiceGUI con sotto FastAPI\Uvicorn e Socket.IO per il backend e Quasar con Tailwind per il front-end. Le varie task vengono coordinate con asyncio per evitare chiamate bloccanti. Il backend si appoggia a pyVISA per le connessioni con strumenti di misura per superare le limitazioni date da interfacce differenti. I grafici vengono renderizzati attraverso Plotly.js.

FILE	RESPONSABILITÀ
main.py	Creazione della User Interface e inizializzazione pannelli di controllo strumenti

FILE	RESPONSABILITÀ
instruments.py	Gestione connessioni con servizi e strumenti.
config.py	Indirizzi strumenti, password, costanti, tabella comandi. Non presente in produzione.

2.2. Dipendenze

Le librerie necessarie sono elencate nei requirements. Le core sono:

```
nicegui>=2.0
pyvisa>=1.14
pyvisa-py>=0.7
numpy>=1.24
```

2.3. Flusso Dati

L'applicativo supporta due modalità di utilizzo: operatore e osservatore.

In modalità osservatore l'interfaccia è passiva: il browser riceve e visualizza i grafici spettrali delle due IF, i dati dell'antenna e la griglia azimut/elevazione senza richiedere credenziali né interazione con gli strumenti. L'applicativo è trasparente, l'osservatore vede i dati in tempo reale senza poter modificare alcun parametro.

In modalità operatore, dopo l'autenticazione, si attivano i controlli di configurazione. E si può, per esempio, avviare il continuous sweep: da quel momento main.py lancia un task dedicato per ciascuna istanza del Panel Analyzer (_background_reader) che

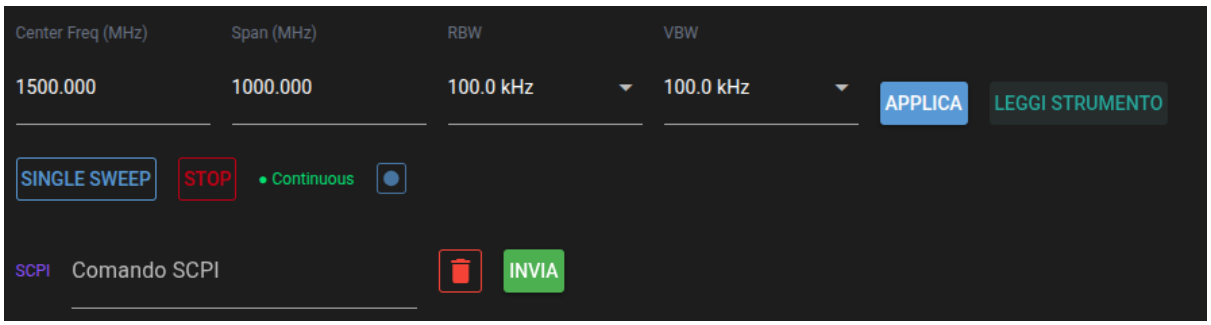
interroga ciclicamente lo strumento tramite InstrumentManager; le tracce vengono lette con una semplice query che restituisce i dati nel seguente formato:

 # 9 000045014 -1.49e+01 XXX....

#	Carattere di inizio blocco
9	Lunghezza campo dedicato al payload
000045014	Payload 45014 byte
-1.49e+01	Valore di ampiezza riportato

I dati acquisiti vengono poi assegnati a dei dizionari globali (scrivibili solo dal processo chiamato)

Le impostazioni che l'operatore può modificare sull'analizzatore al momento sono pressoché illimitate, salvo che si usi il campo "free field".



1) Console Operatore

Sul lato UI, ogni client connesso dispone di un suo timer che scatta all'intervallo configurato in config.py. Il timer chiama la funzione (_refresh_chart) la quale confronta il numero di versione traccia corrente con quello dell'ultimo aggiornamento: solo in

[1] Console operatore

caso di discrepanza il grafico viene ridisegnato, evitando render inutili quando lo sweep time dello strumento è più lungo dell'intervallo di polling. Questo disaccoppiamento consente inoltre a più client di connettersi contemporaneamente condividendo la stessa cache, senza moltiplicare le query VISA verso gli strumenti.

3. SOFTWARE

3.1. InstrumentManager (instruments.py)

La classe InstrumentManager è una singola istanza che gestisce le connessioni verso i due analizzatori. Ogni strumento è rappresentato da un InstrumentHandle che incapsula indirizzo IP, ID strumento e un threading.Lock dedicato. Grazie ai lock separati, le letture dei due analizzatori possono avvenire in parallelo, mentre gli accessi allo stesso strumento sono serializzati.

I metodi principali sono:

- connect(instr_id) / disconnect(instr_id) — apertura e chiusura del socket
- get_params(instr_id) — lettura di center freq, span, RBW, VBW e sweep time
- set_center_freq / set_span / set_rbw / set_vbw — impostazione parametri
- read_trace(instr_id, n) — lettura della traccia <n> (1=CW, 2=MaxHold)
- free_field(instr_id, cmd) — invio di comandi SCPI arbitrari

Il metodo read_trace interroga prima gli estremi di frequenza (:SENSe:FREQuency:STARt? e :STOP?) per costruire l'asse X, poi legge i dati con :TRACe:DATA? TRACE<n>. La risposta contiene un header IEEE 488.2 che viene rimosso prima del parsing.

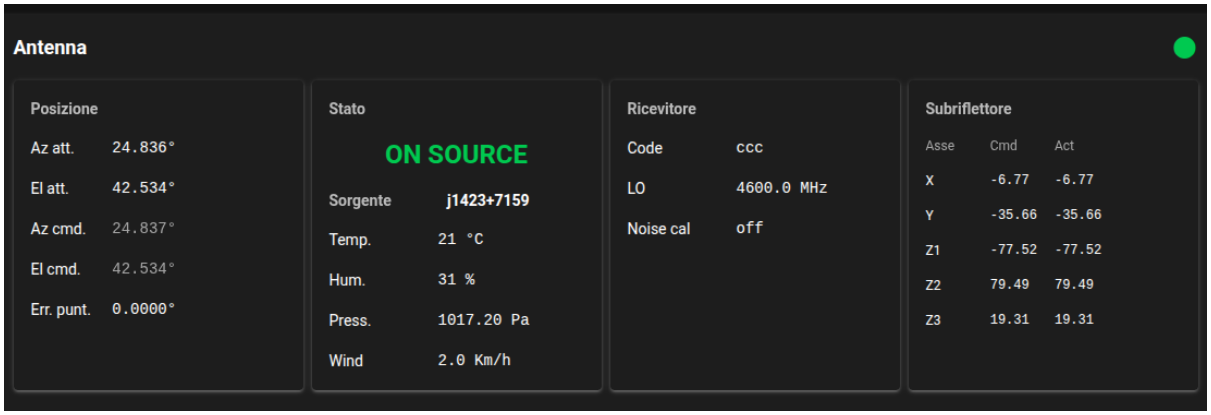
3.2. ServManager (instruments.py)

La classe ServManager gestisce la connessione al servizio serv in esecuzione sulla workstation Field System. La comunicazione avviene tramite un socket TCP con protocollo testuale a riga: il client invia una keyword e il servizio risponde con un array di valori separati da spazi.

Il monitoraggio è suddiviso in due cicli a frequenza diversa:

KEYWORD	FREQUENZA	DATI ACQUISITI
fupdate	Ogni ciclo (2 s)	Az/El attuali e comandati, errore di puntamento, on-source flag
ska	Ogni ciclo (2 s)	Velocità vento
updtrec	Ogni 5 cicli (10 s)	Tipo ricevitore, frequenza LO, noise cal, temperatura, pressione, umidità
updtsub	Ogni 5 cicli (10 s)	Posizioni comandate e attuali dei 5 assi del subriflettore
updsrce	Ogni 5 cicli (10 s)	Nome della sorgente osservata

In caso di disconnessione, ServManager ritenta automaticamente ogni 10 secondi. Lo stato di connessione è esposto tramite il flag ServManager.connected, visualizzato nel pannello antenna tramite un indicatore colorato.



2) Pannello Antenna

4. INTERFACCIA UTENTE

4.1. Layout generale

La pagina principale è raggiungibile all'indirizzo http://192.167.189.31:8080 da qualsiasi browser. L'interfaccia è organizzata verticalmente con i seguenti pannelli, nell'ordine:

1. Barra superiore: titolo e orologio
2. Pannelli Analizzatore: uno per ciascuna IF (IF1 e IF2)
3. Pannello di comparazione: overlay delle tracce per il confronto delle polarizzazioni
4. Pannello Antenna: stato dell'antenna e del ricevitore in tempo reale da Field System

4.2. Autenticazione operatore

L'interfaccia è accessibile in sola lettura senza credenziali: chiunque può visualizzare gli spettri. I controlli di configurazione sono protetti da password e diventano visibili

[2] Pannello Antenna

solo dopo l'autenticazione. La sessione è mantenuta tramite cookie registrati con una chiave segreta definita in config.py.

4.3. Pannello Analizzatore

Ogni pannello analizzatore espone le seguenti funzionalità:

1. Grafico interattivo con Trace 1 ■ e Trace 2 ■
2. Pulsante MaxHold: mostra/nasconde la traccia di massima
3. Pulsante History: sovrappone gli ultimi 8 sweep come tracce grigie sfumate
4. Download CSV: esporta la traccia corrente con metadati
5. Controlli parametri: Center Freq, Span, RBW, VBW con lettura dallo strumento
6. Single Sweep / Continuous Sweep: possibili modalità di acquisizione
7. Registrazione stream: scrive una riga su file csv per sweep in modalità continuous
8. Campo SCPI libero: con autocompletamento su quasi tutti i comandi

Di seguito si offre una descrizione dei parametri immediatamente modificabili dal pannello operatore:

- **Center Freq**: Permette di impostare la frequenza centrale del nostro analizzatore, il valore immesso deve essere in MHz. Potremo dire che è il punto centrale del nostro grafico.
- **Span**: Indica la larghezza della banda osservata a partire dalla frequenza centrale. Supponendo di voler osservare la banda compresa tra 540 e 580 MHz, e avendo una frequenza centrale di 560 MHz, lo span dovrà essere di 40 MHz.
- **RBW (Resolution BandWidth):** Modifica la risoluzione dell'analizzatore per permettere di distinguere segnali deboli o vicini in frequenza. Il limite inferiore per il nostro analizzatore è di 100Hz, permettendo di distinguere due segnali "distanti" almeno il suddetto valore. Diminuendo la RBW si può

notare anche un abbassamento del Noise Floor, a discapito però dello sweep time che rischia di allungarsi oltre il secondo.

- **VBW (Video BandWidth):** Questo parametro influenza solo la visualizzazione della traccia video. Viene applicato un filtro che, in base all'ampiezza impostata, restituisce una traccia pulita e gradevole.
- **SCPI:** L'uso di questa casella permette di interagire con lo strumento attraverso il protocollo SCPI.

4.4. Pannello Antenna

Il pannello antenna si aggiorna ogni 2 secondi e mostra in forma tabellare:

- azimut e elevazione attuali e comandati.
- errore di puntamento
- stato on-source.
- nome della sorgente osservata.
- dati meteo (temperatura, umidità, pressione, vento)
- tipo di ricevitore, frequenza LO.
- marca di rumore.
- posizioni dei cinque assi del subriflettore.

La sezione espandibile "Posizione Antenna" include un grafico che mostra un diagramma azimut/elevazione che traccia il percorso dell'antenna negli ultimi 180 punti, con il punto corrente evidenziato in rosso.

4.5. Pannello di comparazione

Il pannello di confronto sovrappone le ultime tracce dei due analizzatori sullo stesso grafico, con colori distinti. Calcola e visualizza il delta tra le due tracce, utile per verificare la differenza di guadagno tra le due polarizzazioni del ricevitore o per individuare RFI con un certo grado di approssimazione.

5. REGISTRAZIONE DATI

5.1. Stream CSV (solo operatori)

La funzione di registrazione è attivabile dal pulsante REC durante il continuous sweep. I file vengono salvati nella cartella recordings/ con nome <instr_id>_<UTC_timestamp>.csv. Il file ha un header commentato con le condizioni osservative al momento dell'avvio:

```

# Spettro – <label strumento>
# Start UTC: YYYY-MM-DD HH:MM:SS
# LST: HH:MM:SS
# Az: XXX.XXX°  El: YYY.YYY°
# Freq: F_start–F_stop MHz  Points: N

```

Le righe dati hanno il formato:

utc, az_deg, el_deg, t1_0, t1_1, ..., t1_N, t2_0, t2_1, ..., t2_N

dove t1_<i> e t2_<i> sono rispettivamente le ampiezze in dBm delle tracce per il punto definito. Il file è scritto in modalità line-buffered per garantire che le tracce non vengano interrotte a metà.

5.2. Download CSV istantaneo

In qualsiasi momento è possibile esportare lo snapshot corrente delle tracce tramite il pulsante di download. Il file generato contiene le colonne `freq_mhz`, `trace1_dbm` e, se disponibile, `trace2_dbm`, precedute dagli stessi metadati dell'header di registrazione in continuous.

6. INSTALLAZIONE E CONFIGURAZIONE

6.1. Requisiti

I file necessari all'installazione dell'applicativo si possono trovare al link Mopa Repo.

L'applicazione richiede Python 3.10 o superiore. Le dipendenze si installano con:

```
pip install -r requirements.txt
```

```
O singolarmente seguendo l'elenco:
```

- nicegui>=2.0
- pyvisa>=1.14
- pyvisa-py>=0.7
- numpy>=1.24

Si consiglia di procedere con l'installazione all'interno di un virtual environment o, se si vuole velocizzare il deploy, procedere alla creazione di un container. Al momento il servizio gira all'interno di docker su Debian 12.13.

6.2. Parametri

L'applicativo si appoggia a un file di configurazione denominato config.py per facilitare la lettura e la modifica di parametri e strumentazione.

PARAMETRO	DESCRIZIONE
INSTRUMENTS	Dizionario con indirizzi (TCPIP0:<IP>:INSTR) e label per ciascun analizzatore

PARAMETRO	DESCRIZIONE
SERV_HOST / SERVPORT	IP e porta TCP del servizio serv su Field System
OPERATOR_PASSWORD	Password per sbloccare i controlli di configurazione nell'UI
STORAGE_SECRET	Chiave di cifratura dei cookie di sessione NiceGUI
VISA_TIMEOUT_MS	Timeout per le query SCPI in millisecondi (default: 10 000)
TRACE_UPDATE_INTERVAL_S	Intervallo di polling in modalità continuous sweep (default: 1.0 s)
RECORDINGS_DIR	Percorso della cartella di output per i file di registrazione (default: recordings/)
MEDICINA_LONGITUDE_DEG	Longitudine della stazione per il calcolo del LST (default: 11.6450° E)

6.3. Avvio

In dev env l'applicazione si avvia con il comando:

```
python main.py
```

L'interfaccia sarà disponibile su `http://0.0.0.0:8080`. Per l'accesso remoto, assicurarsi che la porta 8080 sia raggiungibile sulla LAN. L'avvio del monitoraggio antenna (`ServManager`) è automatico; se il Field System non è raggiungibile, il sistema riprova ogni 10 secondi senza interrompere le altre funzioni.

6.4. Dockerize

Se si vuole procedere con il deploy attraverso docker, scaricare i file necessari dalla repo e assicurarsi di lavorare in una directory dove si hanno tutti i permessi. Interagendo con Docker potrebbe essere necessario elevarsi a root, per risolvere questo problema aggiungere il proprio utente al gruppo docker. La soluzione più sicura rimane comunque seguire la procedura di configurazione del docker-rootless in modo da confinare qualsiasi attività all'interno di un dato namespace. Supponendo

che il docker engine sia stato installato è il momento di creare due file che ci aiuteranno a creare il container:

Dockerfile [docs]

Questo file è necessario per creare la nostra immagine e impostare i parametri di base (e.g. quali porte userà il nostro servizio).
```

```
FROM python:3.11-slim-bookworm

MAINTAINER Andrea Atza andrea.atza@inaf.it

WORKDIR <la directory dove vivrà il container>

RUN apt-get update && apt-get install -y --no-install-
recommends \
gcc \
&& rm -rf /var/lib/apt/lists/*

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY ./app .

EXPOSE 8080

CMD ["python", "main.py"]

```

docker-compose.yml - standard attuale compose.yaml

Quando si lavora con tanti container o si ha bisogno di specificare diversi servizi il "compose" diventa indispensabile. Ogni parametro definito in compose può essere passato da riga al Docker ma diventerebbe un esercizio di dattilografia. In questo momento dovremo pensare anche alla creazione di un file .env per i nostri dati sensibili e di un dockerignore per evitare di portarci dentro il container anche tutta la parte di sviluppo.
```

Nel “listato” sotto riportato andiamo a creare un servizio {app} che cerca un Dockerfile nella sua stessa directory, builda l’immagine, e poi aggiunge o sovrascrive alcuni parametri. Al nostro servizio in questo caso vengono mappate anche delle directory e file locali. In mancanza di specifiche per la sezione “networks” Docker crea un bridge sulla rete dell’host, per quanto non sia indicato in termini di sicurezza facilita di molto il deploy in ambienti di sviluppo.

```
services:
  app:
    build: .
    container_name: nicegui-app
    ports:
      - "<IP>:<PORT>:<PORT>"
    volumes:
      - ./recordings:/app/recordings
      - ./logs:/app/logs
      - /etc/localtime:/etc/localtime:ro
    restart: unless-stopped
    environment:
      - PYTHONUNBUFFERED=1
    env_file:
      - .env
```

Adesso che abbiamo tutti e due i nostri file ci basterà procedere scrivendo

```
docker compose up --build -d
```

```
Vedremo la nostra immagine che viene creata e poi il processo
si staccherà andando in background.
```

8. RIFERIMENTI

[1] Rigol Technologies, "DSA1000 Series Spectrum Analyzer Programming Guide"

[2] NiceGUI Project, "NiceGUI Documentation", https://nicegui.io.

[3] PyVISA Authors, "PyVISA Documentation", https://pyvisa.readthedocs.io.

APPENDICE A — `instruments.py`

```
"""

InstrumentManager

"""

from __future__ import annotations

import asyncio

import socket

import numpy as np

import pyvisa

import logging

import threading

from dataclasses import dataclass, field

from typing import Optional

from config import INSTRUMENTS, VISA_TIMEOUT_MS

logger = logging.getLogger(__name__)

reader: asyncio.StreamReader

writer: asyncio.StreamWriter

# Data classes per stato antenna da serv.c

@dataclass

class AntennaState:

az_deg: float = 0.0

el_deg: float = 0.0

az_cmd: float = 0.0

el_cmd: float = 0.0

point_err: float = 0.0
```

```
on_source: int = 0

name_src: str = "?"

rx_type: str = "?"

lo_mhz: float = 0.0

noise_cal: int = 0

sub_act: list[float] = field(default_factory=lambda: [0.0] * 5)

sub_cmd: list[float] = field(default_factory=lambda: [0.0] * 5)

sub_mode: int = 0

temp: float = 0.0

pres: float = 0.0

hum: float = 0.0

wind: float = 0.0

#  Data classes per i risultati

@dataclass

class TraceData:

"""Risultato di una lettura traccia."""

frequencies: np.ndarray # asse X in Hz

amplitudes: np.ndarray # asse Y in dBm

start_freq: float

stop_freq: float

num_points: int

@dataclass

class InstrumentParams:

"""Stato corrente dei parametri dell'analizzatore."""

center_freq_hz: float = 0.0
```

```
span_hz: float = 0.0

start_freq_hz: float = 0.0

stop_freq_hz: float = 0.0

rbw_hz: float = 0.0

vbw_hz: float = 0.0

sweep_time_s: float = 0.0

@dataclass

class InstrumentHandle:

"""Stato interno per singolo strumento."""

visa_address: str

label: str

resource: Optional[pyvisa.resources.MessageBasedResource] = None

lock: threading.Lock = field(default_factory=threading.Lock)

connected: bool = False

class InstrumentManager:

"""

Gestisce le connessioni.

"""

def __init__(self) -> None:

self._rm = pyvisa.ResourceManager("@py") # usa pyvisa-py backend

self._instruments: dict[str, InstrumentHandle] = {}

for instr_id, cfg in INSTRUMENTS.items():

self._instruments[instr_id] = InstrumentHandle(

visa_address=cfg["visa_address"],

label=cfg["label"],
```

```
)

@property

def instrument_ids(self) -> list[str]:

return list(self._instruments.keys())

def get_label(self, instr_id: str) -> str:

return self._instruments[instr_id].label

def is_connected(self, instr_id: str) -> bool:

return self._instruments[instr_id].connected

#  Connessione / Disconnessione

def connect(self, instr_id: str) -> None:

handle = self._instruments[instr_id]

with handle.lock:

if handle.connected:

return

try:

res = self._rm.open_resource(handle.visa_address)

res.timeout = VISA_TIMEOUT_MS

idn = res.query("*IDN?").strip()

logger.info("Connesso a %s: %s", instr_id, idn)

handle.resource = res

handle.connected = True

except pyvisa.Error as exc:

logger.error("Connessione fallita per %s: %s", instr_id, exc)

raise ConnectionError(

f"Impossibile connettersi a {handle.label} ({handle.visa_address})"
```

```
) from exc

def disconnect(self, instr_id: str) -> None:

handle = self._instruments[instr_id]

with handle.lock:

if handle.resource is not None:

try:

handle.resource.close()

except pyvisa.Error:

pass

handle.resource = None

handle.connected = False

logger.info("Disconnesso da %s", instr_id)

def disconnect_all(self) -> None:

for instr_id in self._instruments:

self.disconnect(instr_id)

# Comandi SCPI

def _write(self, handle: InstrumentHandle, cmd: str) -> None:

# Invia un comando SCPI

handle.resource.write(cmd)

def _query(self, handle: InstrumentHandle, cmd: str) -> str:

"""Invia una query SCPI e ritorna la risposta (lock già acquisito)."""

return handle.resource.query(cmd).strip()

def _query_float(self, handle: InstrumentHandle, cmd: str) -> float:

return float(self._query(handle, cmd))

# Lettura parametri
```

```
def get_params(self, instr_id: str) -> InstrumentParams:

"""Legge lo stato corrente dell'analizzatore"""

handle = self._instruments[instr_id]

with handle.lock:

if not handle.connected:

raise ConnectionError(f"{instr_id} non connesso")

return InstrumentParams(

center_freq_hz=self._query_float(handle, ":SENSe:FREQuency:CENTer?"),

span_hz= self._query_float(handle, ":SENSe:FREQuency:SPAN?"),

start_freq_hz= self._query_float(handle, ":SENSe:FREQuency:STARt?"),

stop_freq_hz= self._query_float(handle, ":SENSe:FREQuency:STOP?"),

rbw_hz= self._query_float(handle, ":SENSe:BANDwidth:RESolution?"),

vbw_hz= self._query_float(handle, ":SENSe:BANDwidth:VIDeo?"),

sweep_time_s= self._query_float(handle, ":SENSe:SWEep:TIME?"),

)

# Impostazione parametri

def set_center_freq(self, instr_id: str, freq_hz: float) -> None:

handle = self._instruments[instr_id]

with handle.lock:

self._write(handle, f":SENSe:FREQuency:CENTer {freq_hz:.0f}")

def set_span(self, instr_id: str, span_hz: float) -> None:

handle = self._instruments[instr_id]

with handle.lock:

self._write(handle, f":SENSe:FREQuency:SPAN {span_hz:.0f}")

def set_rbw(self, instr_id: str, rbw_hz: float) -> None:
```

```
handle = self._instruments[instr_id]

with handle.lock:

self._write(handle, f":SENSe:BANDwidth:RESolution {rbw_hz:.0f}")

def set_vbw(self, instr_id: str, vbw_hz: float) -> None:

handle = self._instruments[instr_id]

with handle.lock:

self._write(handle, f":SENSe:BANDwidth:VIDeo {vbw_hz:.0f}")

def set_start_stop(

self, instr_id: str, start_hz: float, stop_hz: float

) -> None:

handle = self._instruments[instr_id]

with handle.lock:

self._write(handle, f":SENSe:FREQuency:STARt {start_hz:.0f}")

self._write(handle, f":SENSe:FREQuency:STOP {stop_hz:.0f}")

def free_field(self, instr_id: str, command: str) -> str | None:

command = command.strip()

if not command:

return None

handle = self._instruments[instr_id]

with handle.lock:

if "?" in command:

return self._query(handle, command)

else:

self._write(handle, command)

return None
```

```
# Lettura traccia

@staticmethod

def _strip_block_header(raw: str) -> str:

"""

Rimuove l'header NI-488.2

"""

if not raw.startswith("#"):

return raw

n_digits = int(raw[1])

header_len = 2 + n_digits

return raw[header_len:].lstrip()

def read_trace(self, instr_id: str, trace_n: int = 1) -> TraceData:

"""

Legge la traccia corrente dal DSA.

ritorna i dati con header #<N>XXX seguito da valori ASCII separati da virgola.

"""

handle = self._instruments[instr_id]

with handle.lock:

if not handle.connected:

raise ConnectionError(f"{instr_id} non connesso")

start = self._query_float(handle, ":SENSe:FREQuency:STARt?")

stop = self._query_float(handle, ":SENSe:FREQuency:STOP?")

raw = self._query(handle, f":TRACe:DATA? TRACE{trace_n}")

raw = self._strip_block_header(raw)

amplitudes = np.array([float(v) for v in raw.split(",")])
```

```
num_points = len(amplitudes)

frequencies = np.linspace(start, stop, num_points)

return TraceData(

frequencies=frequencies,

amplitudes=amplitudes,

start_freq=start,

stop_freq=stop,

num_points=num_points,

)

class ServManager:

"""

Gestisce la connessione al servizio hostato su Workstation Field System.

Il servizio risponde a richieste TCP con array che descrivono

lo stato corrente dell'antenna (posizione, frequenza, ecc).

"""

state: AntennaState = AntennaState()

connected: bool = False

@staticmethod

async def _query_async(

reader: asyncio.StreamReader,

writer: asyncio.StreamWriter,

keyword: str,

) -> list[str]:

writer.write((keyword + "\n").encode())

await writer.drain()
```

```
line = await asyncio.wait_for(reader.readline(), timeout=5.0)

return line.decode().split()[1:] # scarta prefisso lunghezza

@staticmethod

async def antenna_monitor(

host: str,

port: int,

fast_interval: float = 2.0,

slow_every: int = 5,

):

tick = 0

while True:

try:

reader, writer = await asyncio.open_connection(host, port)

ServManager.connected = True

logger.info("Connesso a serv su %s:%d", host, port)

try:

while True:

await ServManager._update_fast(reader, writer)

if tick % slow_every == 0:

await ServManager._update_slow(reader, writer)

tick += 1

await asyncio.sleep(fast_interval)

finally:

ServManager.connected = False

writer.close()
```

```
await writer.wait_closed()

except (OSError, asyncio.IncompleteReadError, asyncio.TimeoutError) as exc:

ServManager.connected = False

logger.warning("serv disconnesso: %s — retry in 10 s", exc)

tick = 0

await asyncio.sleep(10)

@staticmethod

async def _update_fast(

reader: asyncio.StreamReader,

writer: asyncio.StreamWriter,

) -> None:

# azc elc azp elp daz del pnterr ionsor incyc

# [0] [1] [2] [3] [4] [5] [6] [7] [8]

t = await ServManager._query_async(reader, writer, "fupdate")

ServManager.state.az_cmd = float(t[0])

ServManager.state.el_cmd = float(t[1])

ServManager.state.az_deg = float(t[2])

ServManager.state.el_deg = float(t[3])

ServManager.state.point_err = float(t[6])

ServManager.state.on_source = int(t[7])

w = await ServManager._query_async(reader, writer, "ska")

ServManager.state.wind = float(w[-3])

@staticmethod

async def _update_slow(

reader: asyncio.StreamReader,
```

```
writer: asyncio.StreamWriter,

) -> None:

# ncal rtype lofreq loampl lostat rtemp rvac Temp Press Hum

# [0]  [1]     [2]    [3]    [4]   [5]  [6]   [7]  [8]  [9]

r = await ServManager._query_async(reader, writer, "updtrec")

ServManager.state.noise_cal = int(r[0])

ServManager.state.rx_type = r[1]

ServManager.state.lo_mhz = float(r[2])

ServManager.state.temp = float(r[7])

ServManager.state.pres = float(r[8])

ServManager.state.hum = float(r[9])

# cmd1-5 act1-5 mode rtype scust

# [0..4] [5..9] [10] [11] [12]

s = await ServManager._query_async(reader, writer, "updtsub")

ServManager.state.sub_cmd = [float(s[i]) for i in range(0, 5)]

ServManager.state.sub_act = [float(s[i]) for i in range(5, 10)]

ServManager.state.sub_mode = int(s[10])

v = await ServManager._query_async(reader, writer, "updsrce")

ServManager.state.name_src = v[0]
```

APPENDICE B — `main.py`

```
"""
Spectrum Web – Interfaccia web.
"""
from __future__ import annotations
import asyncio
import errno
import io
import logging
import os
from collections import deque
from datetime import datetime, timezone
import numpy as np
from nicegui import app, run, ui
from config import (
DEFAULT_CENTER_FREQ_HZ,
DEFAULT_RBW_HZ,
DEFAULT_SPAN_HZ,
DEFAULT_VBW_HZ,
FREQ_MAX_HZ,
FREQ_MIN_HZ,
MEDICINA_LONGITUDE_DEG,
OPERATOR_PASSWORD,
RBW_OPTIONS_HZ,
STORAGE_SECRET,
TRACE_UPDATE_INTERVAL_S,
```

```
VBW_OPTIONS_HZ,

SCPI_COMMANDS,

SERVPORT,

SERV_HOST,

SUB_AXES,

RECORDINGS_DIR,

)

from instruments import InstrumentManager, InstrumentParams, TraceData, ServManager

logging.basicConfig(level=logging.INFO,                          format="%(asctime)s
[%(levelname)s] %(message)s")

logger = logging.getLogger(__name__)

logging.getLogger("uvicorn").addFilter(

lambda record: "" not in record.getMessage()

)

# Manager globale

mgr = InstrumentManager()

app.on_shutdown(mgr.disconnect_all)

# Stato globale condiviso tra tutti i client

_latest_traces: dict[str, TraceData | None] = {k: None for k in mgr.instrument_ids}

_trace_versions: dict[str, int] = {k: 0 for k in mgr.instrument_ids}

_latest_traces2: dict[str, TraceData | None] = {k: None for k in mgr.instrument_ids}

_trace2_versions: dict[str, int] = {k: 0 for k in mgr.instrument_ids}

_reading: dict[str, bool] = {k: False for k in mgr.instrument_ids}

_continuous_active: dict[str, bool] = {k: False for k in mgr.instrument_ids}

_bg_tasks: dict[str, asyncio.Task | None] = {k: None for k in mgr.instrument_ids}

HISTORY_SIZE = 8 # sweep conservati in buffer FIFO
```

```
_AZ_EL_HISTORY = 180 # punti massimi conservati nella history del grafico antenna

_TRACK_THRESHOLD_DEG = 1.00 # scarto minimo per registrare un nuovo punto

_ON_SOURCE_MAP: dict[int, tuple[str, str]] = {

0: ("OFF SOURCE", "text-red-500"),

1: ("ON SOURCE", "text-green-500"),

2: ("OFFSET", "text-yellow-500"),

}

def is_authenticated() -> bool:

return app.storage.user.get("authenticated", False)

# Background reader

async def _background_reader(instr_id: str) -> None:

""" Legge le tracce e aggiorna la cache globale."""

while _continuous_active[instr_id]:

if not _reading[instr_id] and mgr.is_connected(instr_id):

_reading[instr_id] = True

try:

trace = await run.io_bound(mgr.read_trace, instr_id, 1)

_latest_traces[instr_id] = trace

_trace_versions[instr_id] += 1

trace2 = await run.io_bound(mgr.read_trace, instr_id, 2)

_latest_traces2[instr_id] = trace2

_trace2_versions[instr_id] += 1

except Exception as exc:

if "timeout" in str(exc).lower() or "VI_ERROR_TMO" in str(exc):

logger.warning("Timeout VISA su %s — strumento non risponde", instr_id)
```

```
else:

logger.warning("Errore lettura traccia %s: %s", instr_id, exc)

finally:

_reading[instr_id] = False

await asyncio.sleep(TRACE_UPDATE_INTERVAL_S)

#  Helper di formattazione

def fmt_freq(hz: float) -> str:

"""Formatta una frequenza in modo leggibile."""

if hz >= 1e9:

return f"{hz / 1e9:.4f} GHz"

if hz >= 1e6:

return f"{hz / 1e6:.3f} MHz"

if hz >= 1e3:

return f"{hz / 1e3:.1f} kHz"

return f"{hz:.0f} Hz"

def freq_options_map(options_hz: list[float]) -> dict[float, str]:

"""Crea un dizionario per i select di RBW/VBW."""

return {v: fmt_freq(v) for v in options_hz}

#  Calcolo tempo siderale

def _calc_lst(utc: datetime) -> float:

"""Local Sidereal Time in hours per Medicina (IAU GMST formula)."""

jd = utc.timestamp() / 86400.0 + 2440587.5 # Julian Date

d = jd - 2451545.0 # giorni da J2000.0

gmst_deg = (280.46061837 + 360.98564736629 * d) % 360.0

return ((gmst_deg + MEDICINA_LONGITUDE_DEG) % 360.0) / 15.0 # ore
```

```
def _fmt_hms(hours: float) -> str:

h = int(hours)

m = int((hours - h) * 60)

s = int(((hours - h) * 60 - m) * 60)

return f"{h:02d}:{m:02d}:{s:02d}"

# Creazione pannello per strumento

class AnalyzerPanel:

"""

Widget NiceGUI per analizzatore di spettro.

Gestisce stato connessione, grafico live, e controlli parametri attraverso manager

"""

def __init__(self, instr_id: str) -> None:

self.instr_id = instr_id

self.label = mgr.get_label(instr_id)

self.chart = None

self.status_icon = None

self.status_label = None

self.connect_btn = None

self.params_container = None

self.continuous_btn = None

self.sweep_status = None

self._displayed_version: int = -1

self._displayed_version2: int = -1

self._show_trace2: bool = False

self.trace2_btn = None
```

```
self._display_timer = None

self._sweep_info_label = None

self._history_btn = None

self._show_history: bool = False

self._history: deque = deque(maxlen=HISTORY_SIZE)

self._sweep_time_s: float = 0.0

self._recording: bool = False

self._record_file: io.TextIOWrapper | None = None

self._rec_btn = None

self._build_ui()

def _build_ui(self) -> None:

with ui.card().classes("w-full"):

# Header con stato connessione

with ui.row().classes("w-full items-center justify-between"):

ui.label(self.label).classes("text-xl font-bold")

with ui.row().classes("items-center gap-2"):

self.status_icon = ui.icon("circle").classes("text-red-500")

self.status_label = ui.label("Disconnesso").classes("text-sm")

self.connect_btn = ui.button(

"Connetti",

on_click=self._toggle_connection,

).props("dense")

ui.separator()

#Grafico spettro

self.chart = ui.plotly(
```

```
{
"data": [
{
"x": [], "y": [],
"type": "scatter", "mode": "lines",
"name": "Trace 1 (CW)",
"line": {"color": "#2196F3", "width": 1.5},
},
{
"x": [], "y": [],
"type": "scatter", "mode": "lines",
"name": "Trace 2 (MaxHold)",
"line": {"color": "#FF9800", "width": 1.5},
"visible": "legendonly",
},
*[
{
"x": [], "y": [],
"type": "scatter", "mode": "lines",
"name": "history",
"line": {"color": "#444444", "width": 0.8},
"visible": False,
"showlegend": False,
"hoverinfo": "skip",
}
```

```
for _ in range(HISTORY_SIZE)

],

],

"layout": {

"title": None,

"xaxis": {"title": "Frequenza (MHz)", "tickformat": ".2f"},

"yaxis": {"title": "Ampiezza (dBm)", "range": [-120, 0]},

"margin": {"l": 60, "r": 20, "t": 10, "b": 50},

"height": 350,

"uirevision": "constant",

"legend": {"x": 0.01, "y": 0.99},

},

}

).classes("w-full")

with ui.row().classes("items-center gap-2 -mt-2 mb-1"):

self.trace2_btn = ui.button(

"MaxHold", on_click=self._toggle_trace2

).props("dense outline size=sm")

self._history_btn = ui.button(

"History", on_click=self._toggle_history

).props("dense outline size=sm")

ui.button(

icon="download", on_click=self._download_csv

).props("dense outline size=sm").tooltip("Scarica CSV")

self._sweep_info_label = ui.label("").classes("text-xs text-gray-500 font-mono ml-
2")
```

```
#  Controlli parametri

self.params_container = ui.column().classes("w-full")

with self.params_container:

self._build_controls()

self.params_container.set_visibility(False)

def _build_controls(self) -> None:

"""Crea i controlli per center freq, span, RBW, VBW."""

with ui.row().classes("w-full flex-wrap gap-4 items-end"):

with ui.column().classes("gap-1"):

ui.label("Center Freq (MHz)").classes("text-xs text-gray-500")

self.center_input = ui.number(

value=DEFAULT_CENTER_FREQ_HZ / 1e6,

format="%.3f",

min=FREQ_MIN_HZ / 1e6,

max=FREQ_MAX_HZ / 1e6,

step=1.0,

).classes("w-36")

with ui.column().classes("gap-1"):

ui.label("Span (MHz)").classes("text-xs text-gray-500")

self.span_input = ui.number(

value=DEFAULT_SPAN_HZ / 1e6,

format="%.3f",

min=0,

max=FREQ_MAX_HZ / 1e6,

step=1.0,
```

```
).classes("w-36")

with ui.column().classes("gap-1"):

ui.label("RBW").classes("text-xs text-gray-500")

self.rbw_select = ui.select(

options=freq_options_map(RBW_OPTIONS_HZ),

value=DEFAULT_RBW_HZ,

).classes("w-36")

with ui.column().classes("gap-1"):

ui.label("VBW").classes("text-xs text-gray-500")

self.vbw_select = ui.select(

options=freq_options_map(VBW_OPTIONS_HZ),

value=DEFAULT_VBW_HZ,

).classes("w-36")

ui.button("Applica", on_click=self._apply_params).props("color=primary dense")

ui.button("Leggi  strumento",  on_click=self._read_params).props("color=secondary
dense flat")

with ui.row().classes("w-full items-center gap-4 mt-2"):

ui.button("Single Sweep", on_click=self._single_sweep).props("dense outline")

self.continuous_btn = ui.button(

"Continuous Sweep", on_click=self._toggle_continuous

).props("dense outline")

self.sweep_status = ui.label("").classes("text-xs text-gray-400")

self._rec_btn = (

ui.button(icon="fiber_manual_record", on_click=self._toggle_recording)

.props("dense outline size=sm")

.tooltip("Avvia / Ferma registrazione")
```

```
)

self._rec_btn.set_enabled(False)

with ui.row().classes("w-full items-center gap-4 mt-2"):

ui.label("SCPI").classes("text-xs text-violet-500")

self.command = ui.input(

label="Comando SCPI",

placeholder="e.g. :SENSe:FREQuency:CENTer?",

autocomplete=list(SCPI_COMMANDS.keys()),

).classes("w-64")

(ui.button(color="red",        on_click=lambda:        self.command.set_value(""),
icon="delete")

.props("dense outline"))

ui.button("Invia", on_click=self._send_scpi).props("color=green dense")

self.scpi_response = ui.label("").classes("text-xs text-green-300 font-mono")

@property

def last_trace(self) -> TraceData | None:

"""Ultima traccia disponibile dalla cache globale."""

return _latest_traces.get(self.instr_id)

# Aggiornamento display (legge da cache, nessun I/O)

async def _refresh_chart(self) -> None:

needs_update = False

version = _trace_versions.get(self.instr_id, 0)

if version != self._displayed_version:

trace = _latest_traces.get(self.instr_id)

if trace is not None:

if self._show_history and self.chart.figure["data"][0]["x"]:
```

```
self._history.appendleft({

"x": list(self.chart.figure["data"][0]["x"]),

"y": list(self.chart.figure["data"][0]["y"]),

})

self._update_ghost_traces()

self._displayed_version = version

self.chart.figure["data"][0]["x"] = (trace.frequencies / 1e6).tolist()

self.chart.figure["data"][0]["y"] = trace.amplitudes.tolist()

self._update_sweep_info(num_points=trace.num_points)

needs_update = True

version2 = _trace2_versions.get(self.instr_id, 0)

if version2 != self._displayed_version2:

trace2 = _latest_traces2.get(self.instr_id)

if trace2 is not None:

self._displayed_version2 = version2

self.chart.figure["data"][1]["x"] = (trace2.frequencies / 1e6).tolist()

self.chart.figure["data"][1]["y"] = trace2.amplitudes.tolist()

needs_update = True

if self._recording:

self._write_record_row(trace2)

if needs_update:

self.chart.update()

# Connessione

async def _toggle_connection(self) -> None:

if mgr.is_connected(self.instr_id):
```

```
await self._disconnect()

else:

await self._connect()

async def _connect(self) -> None:

self.connect_btn.props("loading")

try:

await run.io_bound(mgr.connect, self.instr_id)

self._set_connected(True)

await self._read_params()

await self._update_trace()

ui.notify(f"{self.label} connesso", type="positive")

except ConnectionError as exc:

ui.notify(str(exc), type="negative")

finally:

self.connect_btn.props(remove="loading")

async def _disconnect(self) -> None:

self._stop_continuous()

mgr.disconnect(self.instr_id)

self._set_connected(False)

ui.notify(f"{self.label} disconnesso", type="warning")

def _set_connected(self, connected: bool) -> None:

if connected:

self.status_icon.classes(replace="text-green-500")

self.status_label.text = "Connesso"

self.connect_btn.text = "Disconnetti"
```

```
self._displayed_version = -1

self._displayed_version2 = -1

self._display_timer = ui.timer(TRACE_UPDATE_INTERVAL_S, self._refresh_chart)

# stato globale del continuous

if _continuous_active[self.instr_id]:

self.continuous_btn.props("color=negative")

self.continuous_btn.text = "Stop"

self.sweep_status.text = "● Continuous"

self.sweep_status.classes(replace="text-xs text-green-400")

else:

self.status_icon.classes(replace="text-red-500")

self.status_label.text = "Disconnesso"

self.connect_btn.text = "Connetti"

if self._display_timer:

self._display_timer.cancel()

self._display_timer = None

self._update_controls_visibility()

def _update_controls_visibility(self) -> None:

"""Aggiorna visibilità dei controlli in base a connessione e autenticazione."""

auth = is_authenticated()

self.connect_btn.set_visibility(auth)

self.params_container.set_visibility(auth and mgr.is_connected(self.instr_id))

def apply_auth(self, authenticated: bool) -> None:

"""Chiamato dalla pagina quando lo stato di autenticazione cambia."""

self._update_controls_visibility()
```

```
# Sweep

async def _single_sweep(self) -> None:

await self._update_trace()

async def _toggle_continuous(self) -> None:

if _continuous_active[self.instr_id]:

self._stop_continuous()

else:

self._start_continuous()

def _start_continuous(self) -> None:

_continuous_active[self.instr_id] = True

if _bg_tasks[self.instr_id] is None or _bg_tasks[self.instr_id].done():

_bg_tasks[self.instr_id] = asyncio.ensure_future(

_background_reader(self.instr_id)

)

self.continuous_btn.props("color=negative")

self.continuous_btn.text = "Stop"

self.sweep_status.text = "● Continuous"

self.sweep_status.classes(replace="text-xs text-green-400")

self._rec_btn.set_enabled(True)

def _stop_continuous(self) -> None:

_continuous_active[self.instr_id] = False # il task esce dal loop al prossimo ciclo

self.continuous_btn.props(remove="color")

self.continuous_btn.text = "Continuous Sweep"

self.sweep_status.text = ""

self.sweep_status.classes(replace="text-xs text-gray-400")
```

```
if self._recording:

self._stop_recording()

self._rec_btn.set_enabled(False)

# Single sweep

async def _update_trace(self) -> None:

"""Legge TRACE1 e TRACE2 una volta e aggiorna la cache"""

if not mgr.is_connected(self.instr_id) or _reading[self.instr_id]:

return

_reading[self.instr_id] = True

try:

trace: TraceData = await run.io_bound(mgr.read_trace, self.instr_id, 1)

_latest_traces[self.instr_id] = trace

_trace_versions[self.instr_id] += 1

trace2: TraceData = await run.io_bound(mgr.read_trace, self.instr_id, 2)

_latest_traces2[self.instr_id] = trace2

_trace2_versions[self.instr_id] += 1

except Exception as exc:

if "timeout" in str(exc).lower() or "VI_ERROR_TMO" in str(exc):

ui.notify(f"{self.label}: timeout — strumento non risponde", type="negative")

logger.warning("Errore lettura traccia %s: %s", self.instr_id, exc)

finally:

_reading[self.instr_id] = False

def _toggle_trace2(self) -> None:

self._show_trace2 = not self._show_trace2

self.chart.figure["data"][1]["visible"]   =   True   if   self._show_trace2   else
"legendonly"
```

```
self.chart.update()

if self._show_trace2:

self.trace2_btn.props("color=orange")

else:

self.trace2_btn.props(remove="color")

def _toggle_history(self) -> None:

self._show_history = not self._show_history

if self._show_history:

self._history_btn.props("color=purple")

else:

self._history.clear()

self._update_ghost_traces()

self.chart.update()

self._history_btn.props(remove="color")

def _update_ghost_traces(self) -> None:

for i, hist in enumerate(self._history):

self.chart.figure["data"][2 + i]["x"] = hist["x"]

self.chart.figure["data"][2 + i]["y"] = hist["y"]

self.chart.figure["data"][2 + i]["visible"] = True

for i in range(len(self._history), HISTORY_SIZE):

self.chart.figure["data"][2 + i]["x"] = []

self.chart.figure["data"][2 + i]["y"] = []

self.chart.figure["data"][2 + i]["visible"] = False

def _update_sweep_info(self, sweep_time: float | None = None, num_points: int |
None = None) -> None:

parts = []
```

```
if num_points is not None:

parts.append(f"{num_points} pts")

t = sweep_time if sweep_time is not None else self._sweep_time_s

if t > 0:

parts.append(f"{t:.2f} s")

self._sweep_info_label.set_text(" ".join(parts))

def _download_csv(self) -> None:

"""Scarica le tracce correnti come CSV con metadati"""

trace = _latest_traces.get(self.instr_id)

if trace is None:

ui.notify("Nessuna traccia disponibile", type="warning")

return

trace2 = _latest_traces2.get(self.instr_id)

now = datetime.now(timezone.utc)

lst = _calc_lst(now)

s = ServManager.state

buf = io.StringIO()

buf.write(f"# Spettro – {self.label}\n")

buf.write(f"# UTC: {now.strftime('%Y-%m-%d %H:%M:%SZ')}\n")

buf.write(f"# LST: {_fmt_hms(lst)}\n")

buf.write(f"# Az: {s.az_deg:.3f}° El: {s.el_deg:.3f}°\n")

buf.write(f"# Points: {trace.num_points} Sweep: {self._sweep_time_s:.2f} s\n")

buf.write("freq_mhz,trace1_dbm")

if trace2 is not None:

buf.write(",trace2_dbm")
```

```
buf.write("\n")

for i, (f, a) in enumerate(zip(trace.frequencies / 1e6, trace.amplitudes)):

row = f"{f:.6f},{a:.4f}"

if trace2 is not None and i < len(trace2.amplitudes):

row += f",{trace2.amplitudes[i]:.4f}"

buf.write(row + "\n")

filename = f"spectrum_{self.instr_id}_{now.strftime('%Y%m%dT%H%M%SZ')}.csv"

ui.download(buf.getvalue().encode(), filename)

#  Registrazione stream

def _toggle_recording(self) -> None:

if self._recording:

self._stop_recording()

else:

self._start_recording()

def _start_recording(self) -> None:

now = datetime.now(timezone.utc)

try:

os.makedirs(RECORDINGS_DIR, exist_ok=True)

path = os.path.join(

RECORDINGS_DIR,

f"{self.instr_id}_{now.strftime('%Y%m%dT%H%M%SZ')}.csv",

)

self._record_file = open(path, "w", buffering=1) # line-buffered

except OSError as exc:

ui.notify(f"Impossibile aprire il file: {exc}", type="negative")
```

```
return

trace = _latest_traces.get(self.instr_id)

s = ServManager.state

lst = _calc_lst(now)

n = trace.num_points if trace else 601

t1_cols = ",".join(f"t1_{i}" for i in range(n))

t2_cols = ",".join(f"t2_{i}" for i in range(n))

self._record_file.write(f"# Spettro – {self.label}\n")

self._record_file.write(f"# Start UTC: {now.strftime('%Y-%m-%d %H:%M:%S')}\n")

self._record_file.write(f"# LST: {_fmt_hms(lst)}\n")

self._record_file.write(f"# Az: {s.az_deg:.3f}° El: {s.el_deg:.3f}°\n")

if trace is not None:

self._record_file.write(

f"# Freq: {trace.start_freq/1e6:.3f}–{trace.stop_freq/1e6:.3f} MHz"

f" Points: {trace.num_points}\n"

)

self._record_file.write(f"utc,az_deg,el_deg,{t1_cols},{t2_cols}\n")

self._recording = True

self._rec_btn.props("color=negative")

ui.notify("Registrazione avviata", type="positive")

def _stop_recording(self, notify: bool = True) -> None:

self._recording = False

if self._record_file:

try:

self._record_file.close()
```

```
except OSError:

pass

self._record_file = None

self._rec_btn.props(remove="color")

if notify:

ui.notify("Registrazione terminata", type="info")

def _write_record_row(self, trace2: TraceData) -> None:

"""Scrive una riga nel file di registrazione, chiamato ad ogni nuovo sweep"""

trace1 = _latest_traces.get(self.instr_id)

if trace1 is None or self._record_file is None:

return

try:

now = datetime.now(timezone.utc)

s = ServManager.state

row = f"{now.strftime('%Y-%m-%dT%H:%M:%SZ')}"

row += f",{s.az_deg:.3f},{s.el_deg:.3f}"

row += "," + ",".join(f"{v:.4f}" for v in trace1.amplitudes)

row += "," + ",".join(f"{v:.4f}" for v in trace2.amplitudes)

self._record_file.write(row + "\n")

except OSError as exc:

if exc.errno == errno.ENOSPC:

ui.notify("Disco pieno — registrazione interrotta", type="negative")

else:

ui.notify(f"Errore scrittura: {exc}", type="negative")

self._stop_recording(notify=False)
```

```
#  Parametri

async def _apply_params(self) -> None:

if not mgr.is_connected(self.instr_id):

ui.notify("Strumento non connesso", type="negative")

return

try:

await  run.io_bound(mgr.set_center_freq,  self.instr_id,  self.center_input.value  *
1e6)

await run.io_bound(mgr.set_span, self.instr_id, self.span_input.value * 1e6)

await run.io_bound(mgr.set_rbw, self.instr_id, self.rbw_select.value)

await run.io_bound(mgr.set_vbw, self.instr_id, self.vbw_select.value)

ui.notify("Parametri applicati", type="positive")

except Exception as exc:

ui.notify(f"Errore: {exc}", type="negative")

async def _read_params(self) -> None:

if not mgr.is_connected(self.instr_id):

return

try:

p: InstrumentParams = await run.io_bound(mgr.get_params, self.instr_id)

self.center_input.value = p.center_freq_hz / 1e6

self.span_input.value = p.span_hz / 1e6

self.rbw_select.value = p.rbw_hz

self.vbw_select.value = p.vbw_hz

self._sweep_time_s = p.sweep_time_s

self._update_sweep_info(sweep_time=p.sweep_time_s)

except Exception as exc:
```

```
ui.notify(f"Errore lettura parametri: {exc}", type="negative")

async def _send_scpi(self) -> None:

if not mgr.is_connected(self.instr_id):

ui.notify("Strumento non connesso", type="negative")

return

cmd = self.command.value.strip()

if not cmd:

return

try:

result = await run.io_bound(mgr.free_field, self.instr_id, cmd)

self.scpi_response.text = f"→ {result}" if result is not None else "→ OK"

ui.notify("Comando inviato", type="positive")

except Exception as exc:

self.scpi_response.text = ""

ui.notify(f"Errore SCPI: {exc}", type="negative")

# Antenna Panel

class AntennaPanel:

def __init__(self):

self._track_az: deque[float] = deque(maxlen=_AZ_EL_HISTORY)

self._track_el: deque[float] = deque(maxlen=_AZ_EL_HISTORY)

with ui.card().classes("w-full"):

with ui.row().classes("items-center justify-between w-full"):

ui.label("Antenna").classes("text-lg font-bold")

self._status_dot = ui.icon("circle", size="sm").classes("text-red-500")

with ui.grid(columns=4).classes("w-full gap-2"):
```

```
with ui.card().classes("col-span-1"):
ui.label("Posizione").classes("font-bold text-sm opacity-70")
with ui.grid(columns=2):
ui.label("Az att.")
self._az = ui.label("—").classes("font-mono")
ui.label("El att.")
self._el = ui.label("—").classes("font-mono")
ui.label("Az cmd.")
self._az_cmd = ui.label("—").classes("font-mono opacity-60")
ui.label("El cmd.")
self._el_cmd = ui.label("—").classes("font-mono opacity-60")
ui.label("Err. punt.")
self._pterr = ui.label("—").classes("font-mono")
with ui.card().classes("col-span-1"):
ui.label("Stato").classes("font-bold text-sm opacity-70")
self._onsrc = ui.label("—").classes("text-2xl font-bold text-center w-full")
with ui.grid(columns=2):
ui.label("Sorgente").classes("font-bold text-sm opacity-70")
self._nmsrc = ui.label("—").classes("font-bold text-sm text-end w-full")
ui.label("Temp.")
self._temp = ui.label("—").classes("font-mono")
ui.label("Hum.")
self._hum = ui.label("—").classes("font-mono")
ui.label("Press.")
self._pres = ui.label("—").classes("font-mono")
```

```
ui.label("Wind")

self._wind = ui.label("—").classes("font-mono")

with ui.card().classes("col-span-1"):

ui.label("Ricevitore").classes("font-bold text-sm opacity-70")

with ui.grid(columns=2):

ui.label("Code")

self._rx = ui.label("—").classes("font-mono")

ui.label("LO")

self._lo = ui.label("—").classes("font-mono")

ui.label("Noise cal")

self._cal = ui.label("—").classes("font-mono")

with ui.card().classes("col-span-1"):

ui.label("Subriflettore").classes("font-bold text-sm opacity-70")

with ui.grid(columns=3):

ui.label("Asse").classes("text-xs opacity-60")

ui.label("Cmd").classes("text-xs opacity-60")

ui.label("Act").classes("text-xs opacity-60")

self._sub_cmd = []

self._sub_act = []

for i in SUB_AXES:

ui.label(f"{i}").classes("text-xs")

self._sub_cmd.append(ui.label("—").classes("font-mono text-xs"))

self._sub_act.append(ui.label("—").classes("font-mono text-xs"))

with ui.expansion("Posizione Antenna", icon="track_changes").classes("w-full mt-
2"):

self._track_chart = ui.plotly({
```

```
"data": [{
"r": [], "theta": [],
"type": "scatterpolar", "mode": "lines+markers",
"name": "Track",
"line": {"color": "#2196F3", "width": 1.5},
"marker": {"size": [], "color": [], "line": {"color": "white", "width": []}},
"showlegend": False,
}],
"layout": {
"paper_bgcolor": "rgba(0,0,0,0)",
"polar": {
"bgcolor": "rgba(40,40,40,0.5)",
"angularaxis": {
"direction": "clockwise",
"rotation": 90,
"tickmode": "array",
"tickvals": [0, 45, 90, 135, 180, 225, 270, 315],
"ticktext": ["N", "NE", "E", "SE", "S", "SW", "W", "NW"],
"tickfont": {"color": "#cccccc"},
"linecolor": "#555555",
"gridcolor": "#444444",
},
"radialaxis": {
"range": [90, 0],
"tickmode": "array",
```

```
"tickvals": [90, 80, 70, 60, 50, 40, 30, 20, 10, 0],

"ticktext": ["90°", "80°", "70°", "60°", "50°", "40°", "30°", "20°", "10°", "0°"],

"tickfont": {"size": 9, "color": "#aaaaaa"},

"linecolor": "#555555",

"gridcolor": "#444444",

},

},

"margin": {"l": 40, "r": 40, "t": 40, "b": 40},

"height": 500,

"uirevision": "constant",

"font": {"color": "#cccccc"},

},

}).classes("w-full")

ui.timer(2.0, self._refresh)

def _refresh(self):

s = ServManager.state

connected = ServManager.connected

self._status_dot.classes(

remove="text-green-500 text-red-500",

add="text-green-500" if connected else "text-red-500"

)

self._az.set_text(f"{s.az_deg:.3f}°")

self._el.set_text(f"{s.el_deg:.3f}°")

self._az_cmd.set_text(f"{s.az_cmd:.3f}°")

self._el_cmd.set_text(f"{s.el_cmd:.3f}°")
```

```
self._pterr.set_text(f"{s.point_err:.4f}°")

_CAL_COLOR_MAP: dict[int, str] = {

3: "text-blue-400",

4: "text-cyan-400",

}

label, color = _ON_SOURCE_MAP.get(s.on_source, ("?", "text-gray-400"))

if s.on_source == 1 and s.noise_cal in _CAL_COLOR_MAP:

color = _CAL_COLOR_MAP[s.noise_cal]

self._onsrc.set_text(label)

self._onsrc.classes(

remove="text-red-500 text-green-500 text-yellow-500 text-blue-400 text-gray-400",

add=color

)

self._nmsrc.set_text(s.name_src)

self._rx.set_text(s.rx_type)

self._lo.set_text(f"{s.lo_mhz:.1f} MHz")

self._cal.set_text("ON" if s.noise_cal else "off")

self._temp.set_text(f"{s.temp:.0f} °C")

self._hum.set_text(f"{s.hum:.0f} %")

self._pres.set_text(f"{s.pres:.2f} Pa")

self._wind.set_text(f"{s.wind:.1f} Km/h")

for i in range(5):

self._sub_cmd[i].set_text(f"{s.sub_cmd[i]:.2f}")

self._sub_act[i].set_text(f"{s.sub_act[i]:.2f}")

if not self._track_az or (
```

```
(s.az_deg - self._track_az[-1]) ** 2 +

(s.el_deg - self._track_el[-1]) ** 2

) ** 0.5 >= _TRACK_THRESHOLD_DEG:

self._track_az.append(s.az_deg)

self._track_el.append(s.el_deg)

az_list = list(self._track_az)

el_list = list(self._track_el)

n = len(az_list)

sizes = [3] * (n - 1) + [9]

colors = ["#2196F3"] * (n - 1) + ["#FF4444"]

widths = [0] * (n - 1) + [2]

self._track_chart.figure["data"] = [{

"r": el_list, "theta": az_list,

"type": "scatterpolar", "mode": "lines+markers",

"name": "Track",

"line": {"color": "#2196F3", "width": 1.5},

"marker": {"size": sizes, "color": colors, "line": {"color": "white", "width":
widths}},

"showlegend": False,

}]

self._track_chart.update()

# Pannello di confronto polarizzazioni

class ComparisonPanel:

"""Overlay delle tracce analizzatori per confronto polarizzazioni."""

def __init__(self, panels: list[AnalyzerPanel]) -> None:

self.panels = panels
```

```
self.timer = None

with ui.card().classes("w-full"):

with ui.dialog() as self._warn_dialog, ui.card():

ui.label(

'Connettere entrambi gli strumenti in modalità "continuous" '

'per abilitare il confronto'

).classes("p-4")

ui.button("Ok!", on_click=self._warn_dialog.close).props("color=primary")

with ui.row().classes("w-full items-center justify-between"):

ui.label("Confronto Polarizzazioni").classes("text-xl font-bold")

with ui.row().classes("items-center gap-2"):

ui.button("Single", on_click=self._single_update)

self.cont_btn = ui.button(

"Continuous", on_click=self._toggle_continuous

).props("dense outline")

ui.separator()

colors = ["#2196F3", "#FF9800"]

self.chart = ui.plotly({

"data": [

{

"x": [], "y": [],

"type": "scatter", "mode": "lines",

"name": p.label,

"line": {"color": colors[i], "width": 1.5},

}
```

```
for i, p in enumerate(panels)

],

"layout": {

"xaxis": {"title": "Frequenza (MHz)", "tickformat": ".2f"},

"yaxis": {"title": "Ampiezza (dBm)", "range": [-120, 0]},

"margin": {"l": 60, "r": 20, "t": 10, "b": 50},

"height": 400,

"uirevision": "constant",

"legend": {"x": 0.01, "y": 0.99},

},

}).classes("w-full")

self.delta_label = ui.label("").classes("text-xs text-gray-400 font-mono")

async def _single_update(self) -> None:

for i, panel in enumerate(self.panels):

trace = panel.last_trace

if trace is None:

continue

self.chart.figure["data"][i]["x"] = (trace.frequencies / 1e6).tolist()

self.chart.figure["data"][i]["y"] = trace.amplitudes.tolist()

self.chart.update()

self._update_delta()

def _update_delta(self) -> None:

t0 = self.panels[0].last_trace

t1 = self.panels[1].last_trace

if t0 is None or t1 is None:
```

```
return

if len(t0.amplitudes) == len(t1.amplitudes):

delta = t0.amplitudes - t1.amplitudes

self.delta_label.text = (

f"mean: {np.mean(delta):+.2f} dBm | "

f"max: {np.max(delta):+.2f} dBm | "

f"min: {np.min(delta):+.2f} dBm"

)

def _toggle_continuous(self) -> None:

if self.timer:

# Stava girando → fermalo

self.timer.cancel()

self.timer = None

self.cont_btn.props(remove="color")

self.cont_btn.text = "Continuous"

else:

# Non stava girando avvialo

self.timer = ui.timer(TRACE_UPDATE_INTERVAL_S, self._single_update)

self.cont_btn.props("color=negative")

self.cont_btn.text = "Stop"

self._warn_dialog.open()

# ── Pagina principale ──────────────────────────────────────────

@ui.page("/")

# dark mode

def index():
```

```
panels: list[AnalyzerPanel] = []

#  Dialog autenticazione

with ui.dialog() as auth_dialog, ui.card().classes("w-80"):

ui.label("Accesso operatore").classes("text-lg font-bold")

pwd_input = ui.input(

label="Password",

password=True,

password_toggle_button=True,

).classes("w-full")

def _confirm_auth() -> None:

if pwd_input.value == OPERATOR_PASSWORD:

app.storage.user["authenticated"] = True

for p in panels:

p.apply_auth(True)

lock_btn.props("icon=lock_open color=positive")

auth_dialog.close()

ui.notify("Accesso operatore attivato", type="positive")

else:

ui.notify("Password errata", type="negative")

pwd_input.value = ""

pwd_input.on("keydown.enter", _confirm_auth)

with ui.row().classes("w-full justify-end gap-2 mt-2"):

ui.button("Annulla", on_click=auth_dialog.close).props("flat dense")

ui.button("Accedi", on_click=_confirm_auth).props("color=primary dense")

def _toggle_auth() -> None:
```

```
if is_authenticated():

app.storage.user["authenticated"] = False

for p in panels:

p.apply_auth(False)

lock_btn.props("icon=lock color=primary")

ui.notify("Accesso operatore revocato", type="info")

else:

auth_dialog.open()

# Header principale

with ui.row().classes("w-full items-center justify-between px-4 py-2"):

ui.label("Spectrum Web – Medicina").classes("text-2xl font-bold")

with ui.row().classes("items-center gap-4"):

lst_label = ui.label("LST: --:--:--").classes("font-mono text-sm")

lock_btn = ui.button(

icon="lock",

on_click=_toggle_auth,

).props("flat dense round color=primary")

#  Layout principale

with ui.column().classes("w-full max-w-7xl mx-auto px-4 gap-6 pb-8"):

AntennaPanel()

for instr_id in mgr.instrument_ids:

panel = AnalyzerPanel(instr_id)

panels.append(panel)

if is_authenticated():

panel.apply_auth(True)
```

```
ComparisonPanel(panels)

ui.timer(

1.0,

lambda: lst_label.set_text(

f"LST: {_fmt_hms(_calc_lst(datetime.now(timezone.utc)))}"

),

)

# Avvio

if __name__ in {"__main__", "__mp_main__"}:

asyncio.get_event_loop().create_task(

ServManager.antenna_monitor(SERV_HOST, SERVPORT)

)

ui.run(

host="0.0.0.0",

port=8080,

title="Spectrum Web - Medicina",

storage_secret=STORAGE_SECRET,

show=False,

reload=True,

)
```