Applicazione web gestione impianti pannelli solari

Mario Rossi

2025-01-28

Sommario

In questo documento viene spiegato come implementare una web app per la gestione di impianti fotovoltaici.

Viene delineata la struttura del progetto in tutte le sue componenti, backend e frontend, inclusi i dati di esempio. Vengono spiegati i vari pacchetti open source utilizzati, e i problemi e le soluzioni adottate durante lo sviluppo del software. Infine vengono esemplificate l'installazione e l'avvio del programma.

Indice

Elenco delle figure

Capitolo 1

Introduzione alla web app

La web app qui presentata è un progetto per il monitoraggio di impianti fotovoltaici. Una serie di microcontrollori che gestiscono sensori può essere in grado di mandare dati opportunamente formati, e l'app è in grado di estrarre dati e andamenti rilevanti e di presentarli in una comoda interfaccia grafica.

Come molte applicazioni web *stateful*, cioè dotate di database, questo sistema è composto di due parti principali: backend e frontend. Il backend si occupa del processamento e memorizzazione di dati, mentre il frontend dell'interfacciamento tra il backend e l'utente. L'utente finale, infatti, interagisce con l'app solo attraverso un web browser.

Capitolo 2

Backend

2.1 Struttura

La parte server, o di backend, è stata divisa in quattro file sorgenti, ed ognuno ha una specifica funzione:

- classes_orm.py: modelli database
- app.py: API implementata con Flask
- json_http_schema.py: da JSON, a validazione dati, a database
- utils.py: utilità database

Nei prossimi punti vedremo nel dettaglio come sono stati progettati questi componenti, quali software sono stati scelti e perché, e, infine, come questi elementi comunicano fra di loro.

2.2 Strumenti utilizzati

2.2.1 Python e le dipendenze

Il progetto è stato sviluppato con Python 3.11 e testato sia in ambiente Debian GNU/Linux, sia Windows. Per Windows si è verificata la funzionalità con Python 3.12.

Tutte le dipendenze del progetto sono open source e vengono specificate in un file di requisiti. Lo standard di fatto per Python per quanto riguarda i pacchetti è il programma pip. Per convenzione il file con i requisiti è `requirements.txt`. Solitamente le dipendenze di un progetto vengono *pinnate*, cioè viene scelta una versione specifica, oppure un ventaglio di versioni possibili. Questo garantisce una parziale riproducibilità[1] in ambienti diversi.

L'omissione della versione *pinned* implica che si utilizzi sempre l'ultima versione al momento del setup, come avviene in questo progetto:

```
Flask
Flask-APScheduler
Flask-SQLAlchemy
Flask-HTTPAuth
mashumaro
SQLAlchemy
httpx
```

Il motivo è che qui siamo più interessati alle funzionalità generali rispetto alla riproducibilità.

[1] Reproducible builds - Wikipedia: https://en.wikipedia.org/wiki/Reproducible_builds

2.2.2 Flask

Flask[2] è framework Python per creare applicazioni web in modo semplice e che siano in grado di mantenere un livello di semplicità anche mentre si aggiungono nuove funzionalità ad esse.

È una valida e più leggera alternativa a Django[3]. Difatti non contiene alcune utilità, quali ad esempio un admin o un sistema di interfacciamento per i database, però il setup e la curva di apprendimento sono più facili.

Si può dire che Flask in questo progetto giochi un ruolo di *colla* fra tutti gli altri componenti come vedremo in seguito.

2.2.3 SQLAlchemy

SQLAlchemy[4] è un sistema ORM (*Object-relational mapping*) che permette di interagire con database relazionali attraverso oggetti Python. Questo semplifica la gestione delle operazioni sul database, come per esempio la gestione delle connessioni e le relazioni tra tabelle.

Vediamo questi esempi di query tratti dal progetto sia con SQLAlchemy ORM, sia direttamente con SQLite Python, rispettivamente. Incominciamo con SQLAlchemy:

```
reading_class = classes_orm.PlantBatterySystemSensorReading
latest_reading_seconds: int = int(request.args.get('latest_reading_seconds'))
cutoff_time = datetime.datetime.now() - datetime.timedelta(seconds=latest_reading_seconds)

# Filtra i valori all'interno del range di tempo richiesto.
data = db.session.query(reading_class).filter(
    reading_class.battery_id == battery_id,
    reading_class.timestamp >= cutoff_time,
    reading_class.timestamp <= datetime.datetime.now()
).all()
```

Questa è invece la versione SQLite:

```
cursor = conn.cursor()

latest_reading_seconds: int = int(request.args.get('latest_reading_seconds'))
cutoff_time = datetime.datetime.now() - datetime.timedelta(seconds=latest_reading_seconds)

# Raw SQL equivalent of the SQLAlchemy query
sql_query = """
SELECT *
FROM PlantBatterySystemSensorReading
WHERE battery_id = ?
AND timestamp >= ?
AND timestamp <= ?
"""

results = cursor.execute(sql_query, (battery_id, cutoff_time,
↪  datetime.datetime.now())).fetchall()

conn.close()
```

Come si può vedere, lavorare con l'ORM SQLAlchemy implica l'utilizzo di oggetti Python. Il metodo

[2] Welcome to Flask — Flask Documentation: https://flask.palletsprojects.com/en/stable/
[3] The web framework for perfectionists with deadlines | Django: https://www.djangoproject.com/
[4] SQLAlchemy - The Database Toolkit for Python: https://www.sqlalchemy.org/

filter, in questo caso, è equivalente al `WHERE` dell'SQL. In linea di massima il primo metodo è più intuitivo, non è limitato al solo SQLite, e non necessita della gestione di *cursori* database per ogni transazione. Infine, utilizzare pure stringhe per l'interazione può diventare complesso e prone ad errori. Per tutti questi motivi si devono preferire sistemi più astratti come SQLAlchemy.

SQLAlchemy, essendo un pacchetto software *standalone*, può essere integrato in qualunque progetto Python, non necessariamente Flask.

2.3 File sorgenti

2.3.1 classes_orm.py

Il file principale dove è definita la struttura completa del database è `classes_orm.py`. Come detto precedentemente, si utilizza SQLAlchemy per definire tutte le tabelle.

Nei prossimi paragrafi andremo a vedere nel dettaglio la struttura del database usando un approccio *top-down*: partiremo cioè dagli elementi principali per poi arrivare a quelli secondari. Verranno omesse alcune tabelle banali.

Tutte le figure qui presenti sono state generate direttamente dal codice Python usando un programma chiamato eralchemy[5].

2.3.1.1 Classi principali

2.3.1.1.1 *Plant* - Impianto L'oggetto *Plant*, cioè impianto, è alla base del progetto. Ogni impianto deve avere un sistema modulo, un sistema batterie, ed alcuni dati di produzione. Vedremo questi tre elementi di seguito.

Gli impianti sono identificati da un numero di serie (*UUID*) unico e casuale. Ovviamente sono provvisti anche di un nome per rendere facile l'identificazione di questi da parte degli utenti.

Altro aspetto meno importante è che un impianto può avere più di un proprietario.

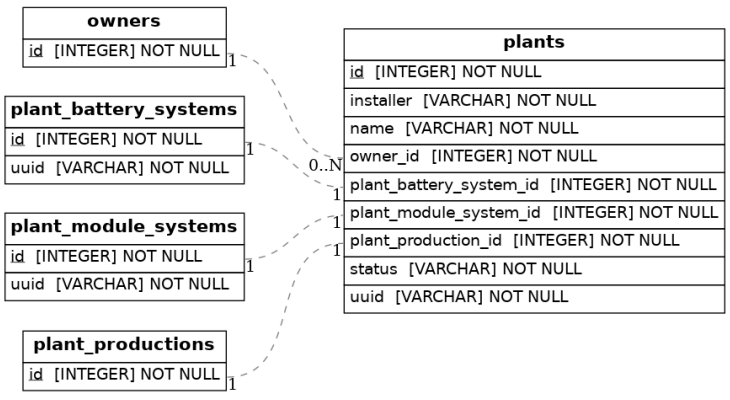

Figura 2.1: Diagramma ER *Plant*

2.3.1.1.2 *PlantModuleSystem* - Modulo sistema impianto Il *PlantModuleSystem*, cioè modulo sistema impianto, contiene le informazioni concernenti i sistemi di produzione di corrente, quindi le letture dei dati dei sensori, e gli intervalli accettabili di questi. Ogni modulo ha una parte a corrente continua ed una alternata.

[5] GitHub - eralchemy/eralchemy: Entity Relation Diagrams generation tool: https://github.com/eralchemy/eralchemy

PlantModuleSystem funge come tabella database intermedia.

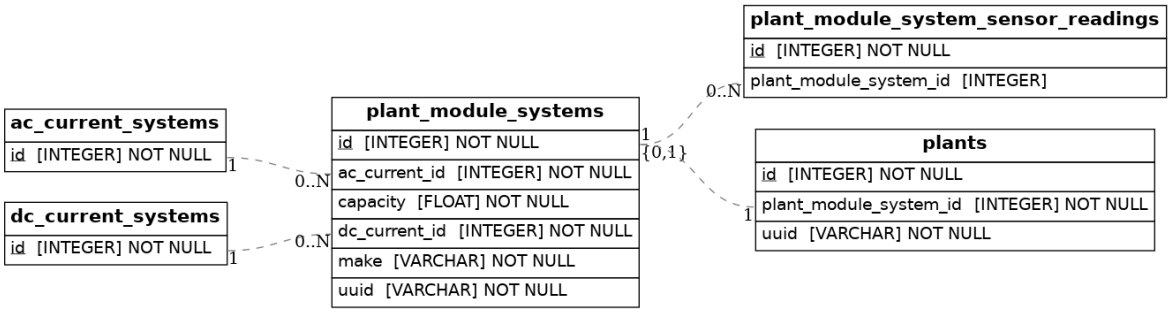

Figura 2.2: Diagramma ER *PlantModuleSystem*

2.3.1.1.3 *PlantBatterySystem* - Sistema batterie impianto Il *PlantBatterySystem*, cioè sistema batterie impianto, è per certi versi simile al *PlantModuleSystem*, ma si occupa dei dati delle batterie di immagazzinamento. In questo caso i range ed i dati delle letture sono legati alle singole batterie e non direttamente al sistema batterie.

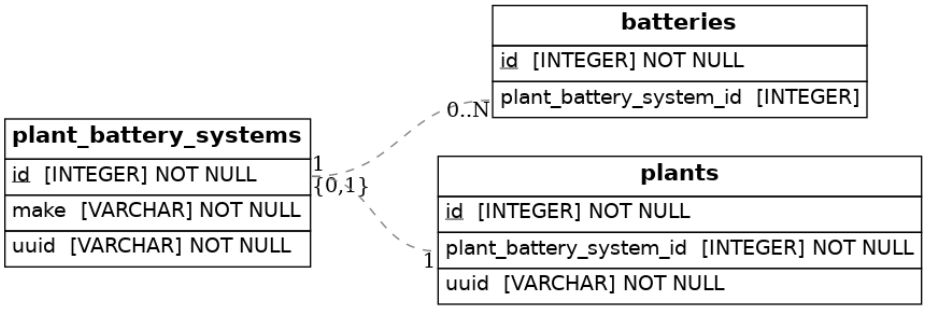

Figura 2.3: Diagramma ER *PlantBatterySystem*

2.3.1.1.4 *Battery* - Batterie Tabella che rappresenta le singole batterie. Ogni impianto ha normalmente due batterie: *master* e *slave*. In modo simile al *PlantModuleSystem*, abbiamo le specifiche delle batterie e le letture dei loro sensori come tabelle collegate.

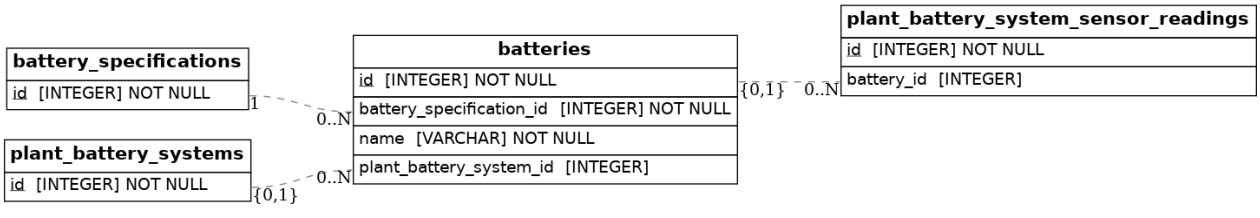

Figura 2.4: Diagramma ER *Battery*

2.3.1.1.5 *PlantModuleSystemSensorReading* - Letture sensori modulo impianto Le letture dei sensori per la parte di produzione di corrente sono gestite dalla tabella

PlantModuleSystemSensorReading. Per letture di corrente continua il valore di frequenza *frequency* è sempre fissato a zero, mentre per la corrente alternata avrà un altro valore a virgola mobile positivo.

Ogni lettura ha un *timestamp*, cioè un elemento temporale corrispondente al momento della misurazione.

Le letture hanno anche un codice di allarme associato. Di norma questo valore sarà `-01`, cioè nessun allarme. Tutti i possibili valori sono definiti in questa enumerazione:

```
class AlarmCode(Enum):
    NO_PROBLEM = '-01'
    SOFTWARE_PROBLEM = '00'
    DC_PROBLEM = '01'
    BMS_PROBLEM = '02'
    SLAVE_BATTERY_PROBLEM = '03'
    PLANT_COMPONENT_PROBLEM = '04'
```

Ogni valore di un'istanza della classe *PlantModuleSystemSensorReading*, inclusi *timestamp* e *alarm_code* sono inviati dai sensori stessi e non gestiti o calcolati dalla web app.

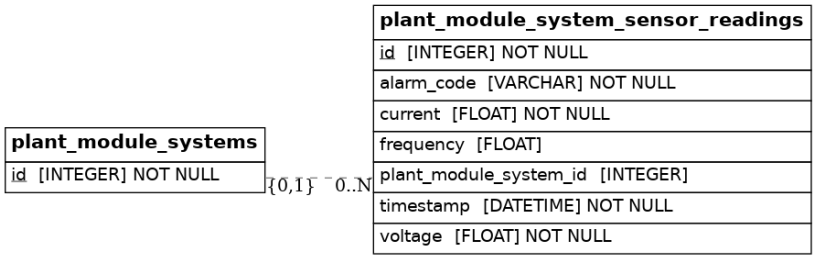

Figura 2.5: Diagramma ER *PlantModuleSystemSensorReading*

2.3.1.1.6 *PlantBatterySystemSensorReading* - Letture sensori modulo batteria In modo simile a *PlantModuleSystemSensorReading*, *PlantBatterySystemSensorReading* gestisce le letture dei sensori di ciascuna batteria, e quindi ogni lettura è associata ad una batteria. Il valore di frequenza, *frequency*, è sempre zero: tutte le batterie sono infatti a corrente continua.

Si è deciso di creare una classe di base per tutte le letture chiamata *SensorReading* e di cambiare alcuni dettagli a seconda del caso d'uso:

```
@dataclass
class SensorReading(Base):
    __abstract__ = True

    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    voltage: Mapped[float] = mapped_column(Float, nullable=False)
    current: Mapped[float] = mapped_column(Float, nullable=False)

    # La frequenza non ha senso per la corrente DC.
    frequency: Mapped[Optional[float]] = mapped_column(Float, default=0.0)

    timestamp: Mapped[datetime] = mapped_column(DateTime(timezone=True), nullable=False)
    alarm_code: Mapped[str] = mapped_column(String, nullable=False, default='-01')
```

La classe *PlantBatterySystemSensorReading* contiene questo metodo di inizializzazione:

```
def __init__(self, voltage: float, current: float, timestamp: datetime.datetime, battery:
↪  Any, frequency: float = 0.0, alarm_code: str = '-01'):
        self.voltage = voltage
        self.current = current

        # La frequenza non ha senso per la corrente DC, quindi anche per le batterie.
        self.frequency = 0.0

        self.timestamp = timestamp
        self.alarm_code = alarm_code

        self.battery = battery
```

Il commento riguardo la frequenza è auto-esplicativo.

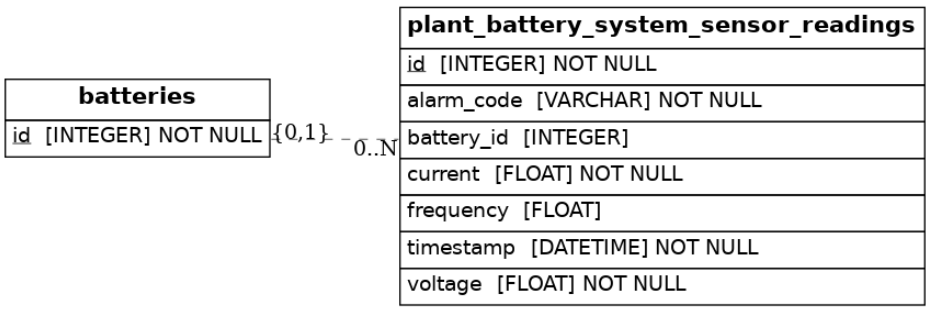

Figura 2.6: Diagramma ER *PlantBatterySystemSensorReading*

2.3.1.1.7 *Alarm* - Allarmi Citati precedentemente, gli allarmi sono una componente fondamentale di questo progetto. Permettono di capire quando e a quale componente si presenta un problema. Ogni allarme ha un codice (*code*), un livello di severità (*severity*), e sono direttamente associati agli impianti ed ai ticket.

Il campo *visible* è strettamente legato ai ticket: quando un ticket è aperto l'allarme è visibile. Una volta che quel ticket viene chiuso i suoi allarmi associati vengono nascosti impostando *visible* = *False*.

Le letture, che siano *PlantModuleSystemSensorReading* oppure *PlantBatterySystemSensorReading*, possono far scattare un allarme sotto determinate condizioni, a partire dai codici di allarme discussi nel punto precedente, così come da letture fuori dai range accettabili.

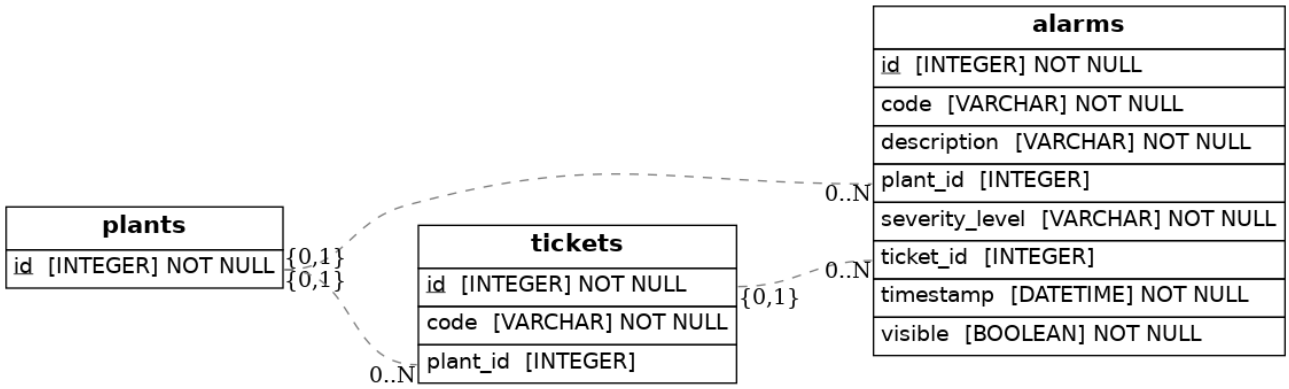

Figura 2.7: Diagramma ER *Alarm*

2.3.1.1.8 *Ticket* - Ticket I ticket rappresentano una serie di problemi sui quali i manutentori o gli sviluppatori devono agire. Ognuno di questi oggetti comprende una raccolta di allarmi e quindi si rifanno agli stessi valori di questi. Possono essere gestiti direttamente dall'utente finale attraverso il frontend ed hanno tre possibili stati:

```
class TicketCode(Enum):
    IN_PROGRESS = 'IN PROGRESS'
    RESOLVED = 'RESOLVED'
    NOT_RESOLVED = 'NOT RESOLVED'
```

L'algoritmo di gestione dei ticket e degli allarmi verrà spiegato nei prossimi punti.

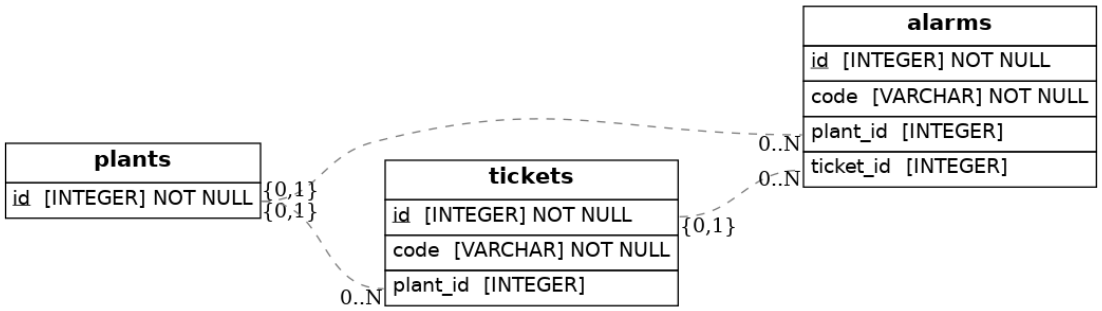

Figura 2.8: Diagramma ER *Ticket*

2.3.1.2 Schema completo

Questo è lo schema relazionale di tutto il progetto.

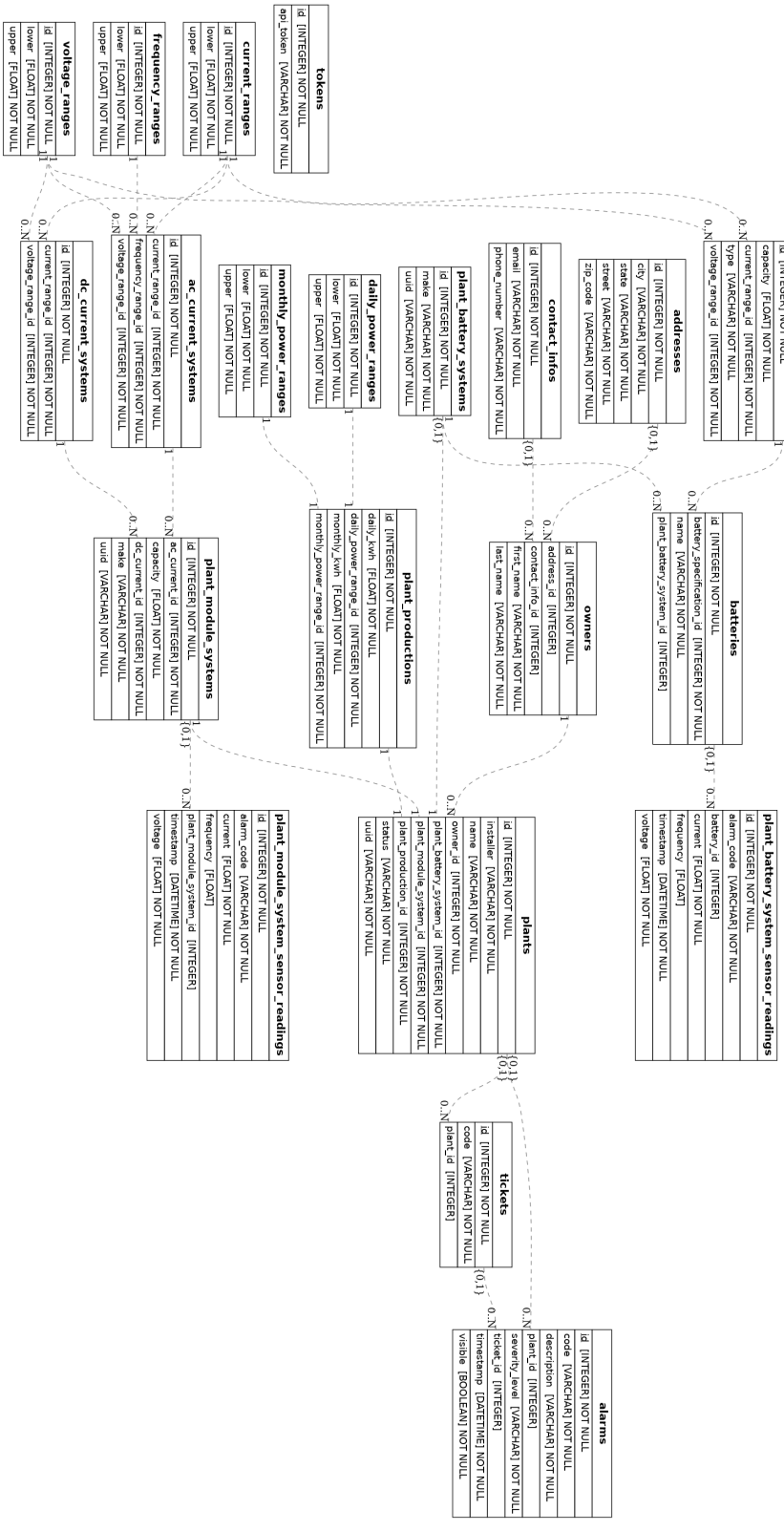

Figura 2.9: Diagramma ER completo

2.3.1.3 Metodi di serializzazione dei dati

Ogni modello, cioè tabella del database, ha una propria funzione di serializzazione dei dati. In questo caso, serializzare significa trasformare oggetti Python in stringhe JSON in modo che le informazioni possano poi essere trasmesse via HTTP.

Ci sono due principali motivi nel definire queste funzioni:

- personalizzazione dell'output
- necessità di evitare ricorsioni infinite

Partiamo da quest'ultimo punto. Quando si usa il metodo `jsonify` direttamente su un oggetto di SQLAlchemy che è stato definito usando le `dataclass`[6], allora si può incappare in ricorsioni infinite[7]. Questo succede perché Python va a iterare su tutti gli attributi, incluse le relazioni tra tabelle. Se ci sono relazioni circolari, cioè collegamenti circolari tra due o più tabelle, l'iterazione non ha un punto di arresto.

Per il secondo punto possiamo analizzare questo esempio tratto dal progetto:

```
def serialize(self) -> dict:
    return {
        'id': self.id,
        'uuid': self.uuid,
        'name': self.name,
        'owner_id': self.owner_id,
        'plant_module_system_id': self.plant_module_system_id,
        'plant_battery_system_id': self.plant_battery_system_id,
        'plant_production_id': self.plant_production_id,
        'installer': self.installer,
        'alarms': [a.id for a in self.alarms],
        'tickets': [t.id for t in self.tickets],
        'status': self.status
    }
```

Questa è la funzione di serializzazione di *Plants*. Possiamo vedere gli attributi semplici e, per quanto riguarda `alarms` e `tickets`, si vanno ad estrarre solo gli ID (numeri interi naturali) degli oggetti associati sotto forma di lista.

Quest'altra funzione è parte della classe *PlantBatterySystemSensorReading*:

```
def serialize(self) -> dict:
    return {
        'id': self.id,
        'voltage': self.voltage,
        'current': self.current,
        'timestamp': self.timestamp.astimezone(datetime.timezone.utc).replace(tzinfo=None).i
        ↪  soformat(timespec='seconds') + 'Z' if self.timestamp else
        ↪  None,
        'frequency': self.frequency,
        'alarm_code': self.alarm_code,
        'battery_id': self.battery_id,
    }
```

L'attributo `timestamp` viene modificato al volo per avere una stringa con il fuso orario UTC, nel

[6] le dataclass sono un modo di definire classi con un insieme di attributi tipizzati (per esempio `str`, `int`, `float` o altre tipi). Di solito vengono utilizzate come alternative per salvare dati rispetto alle classi tradizionali perchè i dati possono essere facilmente trasformati in dizionari. Sono anche presenti metodi come il `__post_init__` e altre caratteristiche.

[7] https://github.com/sqlalchemy/sqlalchemy/discussions/10366#discussioncomment-7059898

formato ISO 8601, con l'aggiunta di `Z` (Zulu) come suffisso. Il frontend sarà poi in grado di decodificare in modo non ambiguo questo valore.

2.3.2 app.py

app.py è il file che Flask va a leggere per primo all'avvio dell'applicazione. Qui è definita tutta la parte di API REST[8], cioè viene messa a disposizione un'interfaccia che fa comunicare gli *endpoint*[9] definiti al suo interno con la parte di database spiegata nei punti precedenti.

Ora vediamo più nel dettaglio come funziona l'API REST in questo progetto.

2.3.2.1 REST API

Per mantenere una struttura semplice ed in linea di massima aderente ai principi REST, ogni tabella del database ha due *viste* possibili all'interno di questo file:

- vista elenco, o lista di elementi
- vista singolo elemento

2.3.2.1.1 Operazioni su liste di elementi La vista ad elenco permette di agire o controllare più oggetti simultaneamente. Prendiamo l'esempio dell'inpianto (*Plant*):

```
@app.route('/plant', methods=['GET', 'POST'])
def rest_plant():
    # Resto del codice.
```

Qui abbiamo definito l'endpoint `/plant`, ed è solo possibile visualizzare tutti gli impianti (`GET`) o aggiungerne uno nuovo (`POST`). I metodi HTTP supportati sono infatti definiti come lista per argomento di `methods`.

Altro dato importante è che, per quanto riguarda `@app.route`, si tratta di un *decoratore*[10] che fa parte di Flask.

2.3.2.1.2 Operazioni su singoli elementi Per agire su un singolo elemento bisogna conoscerne l'identificativo all'interno del database. Ogni tabella definita all'interno del file dei modelli, `classes_orm.py`, ha un campo `id`:

```
id: Mapped[int] = mapped_column(Integer, primary_key=True)
```

Questa variabile deve essere quindi usata per riferirsi al singolo elemento:

```
@app.route('/plant/<int:plant_id>', methods=['GET', 'PUT'])
def rest_show_plant(plant_id):
    # Resto del codice.
```

Nel decoratore abbiamo quindi ora `<int:plant_id>`: si tratta di una variabile Python di tipo `int`, il cui nome `plant_id` viene passato direttamente alla funzione `rest_show_plant`, e quindi se ne potrà fare uso al suo interno.

2.3.2.1.3 Esempi di richieste HTTP Esempi di possibili richieste HTTP effettuate con il programma curl[11] possono essere questi:

[8] Representational state transfer - Wikipedia: https://it.wikipedia.org/wiki/Representational_state_transfer

[9] Un endpoint può essere definito come un URL ben definito attraverso il quale è possibile interagire con una o più risorse remote.

[10] un decoratore è una funzione che altera il comportamento di un'altra funzione.

[11] curl is used in command lines or scripts to transfer data: https://curl.se/

```
curl -X GET http://127.0.0.1:8080/plant

curl -X GET http://127.0.0.1:8080/plant/2
```

Queste servono solo per la visualizzazione di informazioni[12]. Per la creazione di nuovi impianti la richiesta ha bisogno di ulteriori parametri.

2.3.2.1.4 Visualizzazione elementi Come accennato nel punto precedente, la lettura di oggetti viene fatta con il verbo HTTP `GET`.

2.3.2.1.4.1 Casi base Per le viste di singoli elementi si applica un semplice filtro alla query del database per ID elemento:

```
@app.route('/plant/<int:plant_id>', methods=['GET', 'PUT'])
def rest_show_plant(plant_id):
    if request.method == 'GET':
        return utils.detail_db_object(classes_orm.Plant, plant_id, db)
```

Per le viste di elementi multipli esiste un caso base:

```
@app.route('/plant', methods=['GET', 'POST'])
def rest_plant():
    if request.method == 'GET':
        return utils.list_db_objects(classes_orm.Plant, db)
```

Il contenuto di queste due funzioni, `utils.detail_db_object` e `utils.list_db_objects` viene esemplificato nei punti successivi.

2.3.2.1.4.2 Casi complessi In casi più complessi è necessario, per esempio, ordinare l'output con un certo criterio prima di mandare la risposta HTTP:

```
@app.route('/ticket', methods=['GET', 'POST'])
def rest_ticket():
    if request.method == 'GET':
        tickets = db.session.query(classes_orm.Ticket).all()

        # Codice qui omesso per motivi di lettura.

        # Ordina i ticket in base al timestamp del primo allarme.
        sorted_tickets = sorted(tickets_with_timestamps, key=lambda x: x[1], reverse=True)

        # Seleziona solo i ticket.
        sorted_ticket_instances = [ticket for ticket, _ in sorted_tickets]
        return jsonify([ticket.serialize() for ticket in sorted_ticket_instances]), 200
```

In questo esempio i dati sono ordinati per timestamp decrescente, in modo che i più recenti siano all'inizio del payload JSON di risposta.

In altri casi, invece, per semplificare le query che il frontend deve fare sono stati aggiunti dei campi supplementari. Vediamo un estratto dall'endpoint `/plant_module_system_sensor_reading`, che serve a presentare i dati delle letture del sistema modulo impianto. Leggere tutte le letture di tutti gli impianti non è molto utile. In questo progetto infatti l'utente potrà visualizzare solo alcune letture

[12]Ci sono un paio di considerazioni da fare che verranno poi spiegate in dettaglio. Per prima cosa il progetto è stato volutamente semplificato. Non è possibile cancellare oggetti e l'aggiornamento di dati di oggetti esistenti è riservato a pochi elementi.

specifiche (parametro `results`) di un impianto alla volta (parametro `plant_id`). È comunque sempre possibile leggere tutte le letture di tutti gli impianti, come da paradigma REST.

Anche in questo caso, parte del codice non è stato riportato:

```
@app.route('/plant_module_system_sensor_reading', methods=['GET', 'POST'])
def rest_plant_module_system_sensor_reading():
    if request.method == 'GET':

        if 'plant_id' in request.args:
            plant_id: int = int(request.args.get('plant_id'))
            plant = utils.detail_raw_db_object(classes_orm.Plant, plant_id, db)

            if plant and plant.plant_module_system:
                if 'results' in request.args:
                    results: int = int(request.args.get('results'))

                    # Letture filtrate per impianto e limitate per numero di
                    # risultati.
                    data =
↪   db.session.query(classes_orm.PlantModuleSystemSensorReading).filter(
                        classes_orm.PlantModuleSystemSensorReading.plant_module_system_id ==
↪   plant.plant_module_system.id).order_by(
                            desc(classes_orm.PlantModuleSystemSensorReading.timestamp)
            ).limit(results).all()

                    # Resto del codice.
```

Gestire questi dati con Python sicuramente riduce la quantità di dati trasmessa ed anche la difficoltà di filtrarli con il JavaScript nella parte di frontend.

Una query di esempio potrebbe essere la seguente:

```
curl -X GET http://127.0.0.1:8080/plant_module_system_sensor_reading?plant_id=2&results=10
```

2.3.2.1.5 Nuovi elementi Per creare nuovi elementi, il paradigma REST consiglia di utilizzare il verbo HTTP `POST`. Questa operazione è più complessa di una semplice lettura.

2.3.2.1.5.1 Casi semplici Quando si crea un oggetto bisogna prima controllare se tutte le relazioni che si andranno a creare con altri oggetti rispettano i requisiti del database. Se riprendiamo lo schema globale del database e analizziamo tabelle quali:

- *CurrentRange*
- *VoltageRange*
- *FrequencyRange*
- *DailyPowerRange*
- *MonthyPowerRange*
- *Token*

ci accorgiamo che non hanno relazioni dirette ad altre tabelle. La creazione di questi elementi è semplice e non prevede particolari controlli, quindi è sufficiente instanziare l'oggetto e salvarlo nel database, come in questo esempio:

```
elif request.method == 'POST':
    # Altro codice.

    return utils.new_db_object(
```

```
        classes_orm.CurrentRange(
            lower=new_data.lower,
            upper=new_data.upper,
        ),
        db
    )
```

Per creare un nuovo range di corrente è quindi sufficiente fare:

```
curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"lower": 10.0, "upper": 18.0}' \
    http://127.0.0.1:8080/current_range
```

2.3.2.1.5.2 Casi complessi Al contrario, creare un nuovo impianto, per esempio, implica che debbano già essere presenti questi dati nel database:

- un proprietario
- un sistema module impianto
- un sistema batteria impianto
- una tabella per i dati di produzione

Se queste quattro informazioni non esistono nel database, la creazione di un nuovo impianto non andrà a buon file. Questo estratto di codice rende bene l'idea:

```
elif request.method == 'POST':
    # Ricerco il proprietario dell'impianto.
    owner = utils.detail_db_object(classes_orm.Owner, new_data.owner_id, db, False, 'owner')
    if owner is None:
        return jsonify({'error': f'ac_current {new_data.owner_id} not found'}), 404

    # Il sistema modulo impianto deve esistere e non essere associato
    # ad altri impianti. Stesso discorso per il sistema batteria impianto
    # e i dati produzione impianto, entrambi omessi qui.
    plant_module_system = db.session.query(classes_orm.PlantModuleSystem).filter(
            classes_orm.PlantModuleSystem.id == new_data.plant_module_system_id,
        ).first()
    is_available: bool = False
        if plant_module_system is not None:
            if plant_module_system.plant is None:
                is_available = True
    if not is_available:
            return jsonify({'error': f'plant_module_system {new_data.plant_module_system_id}
            ↪  not found or already used'}), 404

    # Resto del codice.

    return utils.new_db_object(
        classes_orm.Plant(name=new_data.name, owner=owner, installer=new_data.installer,
            plant_module_system=plant_module_system,
↪   plant_battery_system=plant_battery_system,
            plant_production=plant_production), db
    )
```

Per creare un nuovo impianto, quindi, è necessario passare l'id di questi elementi nel *payload* (corpo)

JSON della richiesta:

```
curl \
    -X POST \
    - d '{"name": "Mario Rossi", "owner_id": 1, "plant_module_system_id": 1,
↪   "plant_battery_system_id": 1,  "plant_production_id": 1, "installer": "ENEL"}' \
    http://127.0.0.1:8080/plant
```

2.3.2.1.6 Aggiornamento L'aggiornamento di dati esistenti è possibile utilizzando il verbo `PUT`, che in questo progetto viene usato molto selettivamente, in particolare per la gestione dei ticket.

In generale, la progettazione di questo tipo di operazioni deve essere fatta in modo simile alla parte della creazione di oggetti, ma usando il metodo `update` una volta che si è individuato l'istanza da aggiornare.

Una menzione particolare da fare è proprio legata agli allarmi ed ai ticket in quanto il meccanismo può risultare complicato da apprendere. Qui verrà spiegato l'algoritmo in una versione semplificata con pseudocodice. Il codice Python si trova comunque nella funzione `new_issues` e viene chiamato ad ogni creazione di letture.

Algorithm 1 Pseudocodice algoritmo gestione allarmi e ticket

$alarms \leftarrow [\,]$
for all *variable* in *reading* **do** ▷ Controllo dei range
  if $variable < range$ **or** $variable > range$ **then**
    $alarm\_code \leftarrow$ `01`
    $alm \leftarrow message, alarm\_code$
    $alarms \leftarrow \text{append}(alarms, alm)$
  end if
end for
if $new\_data.alarm\_code \neq$ `-01` **then** ▷ Controllo dei codici errore inviati all'API
  $alarm\_code \leftarrow new\_data.alarm\_code$
  $alm \leftarrow message, alarm\_code$
  $alarms \leftarrow \text{append}(alarms, alm)$
end if
for all *alm* in *alarms* **do** ▷ Creazione allarmi
  $db \leftarrow \text{add}(alm)$
end for
$a0 \leftarrow$ Filtra gli allarmi non associati a nessun ticket nel database
$a1 \leftarrow$ Raggruppa $a0$ per tipo
$a2 \leftarrow$ Considera solo allarmi di $a1$ dove ce ne è almeno 1 più recente di entro l'ultima ora
$results \leftarrow$ Considera solo se ci sono $\geq$ 2 allarmi di questo tipo in $a2$
$db \leftarrow \text{add}(ticket)$ ▷ Creazione ticket nel database
$db.ticket.alarms \leftarrow results$
$tkts \leftarrow$ db.filter(Ticket)[with((code == `NOT RESOLVED`) and (with(max(alarm.timestamp)) $\leq$ 1 ora))]
for all *tkt* in *tkts* **do** ▷ Chiudi ticket e nascondi loro allarmi se non c'è stata attività recente
  $db.tkt.code \leftarrow$ `SOLVED`
  for all *alm* in $db.tkt.alarms$ **do**
    $alm.visible \leftarrow$ `False`
  end for
end for

2.3.2.1.7 Cancellazione In questo progetto non è stata implementata nessuna operazione di cancellazione, che, secondo il paradigma REST, si implementa con `DELETE`.

2.3.3 utils.py

Questo file Python contiene una serie di funzioni di convenienza quasi tutte legate all'interazione con il database. Durante lo sviluppo del progetto, per mantenere una struttura più semplice all'interno di `app.py`, si è deciso di creare funzioni per l'inserimento, la visualizzazione e la validazione di dati.

2.3.3.1 Creazione nuovi record in database

La funzione `new_db_object` crea un oggetto e ritorna uno stato che può essere di successo se l'inserimento è andato a buon fine, oppure di errore in caso contrario.

Quando un oggetto viene inserito con `POST`, l'applicazione deve ritornare uno stato HTTP `201`, che significa che una nuova risorsa è stata creata. Nei casi di errore viene ritornato il classico errore `500`. In entrambi i casi, comunque, viene sempre restituito un payload JSON, che, può essere una stampa dell'oggetto oppure dell'errore.

Si è dovuto implementare una *retry strategy*, cioè un modo per ritentare l'inserimento di oggetti nel database. Infatti, utilizzando il server Flask di sviluppo ed il database in memoria RAM, può capitare che alcune operazioni di inserimento non vadano a buon fine. Questi due sistemi non sono progettati per essere particolarmente stabili e ottimizzati per operazioni parallele. La *retry strategy* mitiga un po' questo problema, e, in un server di produzione, non dovrebbe essere necessaria.

Vediamo un estratto di codice, qui leggermente condensato per questioni di leggibilità:

```
def new_db_object(obj: Any, db: Any, max_retries: int = 10000, delay: float = 0.1,
↪  return_raw: bool = False) -> tuple:
    attempt: int = 0

    while attempt < max_retries:
        try:
            db.session.add(obj)
            db.session.commit()
            logging.info(f'Object added: {obj.serialize()}')
            db.session.flush()
            if return_raw:
                return obj.serialize()
            else:
                return jsonify({'data': obj.serialize()}), 201
        except (SQLAlchemyError, IntegrityError) as e:
            db.session.rollback()
            attempt += 1
            time.sleep(delay)
        except Exception as e:
            return jsonify({'error': 'Unexpected error: ' + str(e)}), 500

    return jsonify({'error': 'Failed to add object after multiple attempts.'}), 500
```

Semplicemente, quando troppi tentativi di inserimento non vanno a buon fine l'app ritorna un errore `500`. Dopo ogni errore bisogna ricordarsi di fare il *rollback*, cioè riportare il database allo stato precedente la transazione fallita.

2.3.3.2 Validazione JSON

La funzione `validate_json` controlla i payload in input dalle richieste HTTP. Prima di interagire con il database è necessario controllare che questi dati siano conformi alla struttura del database. Se questo non viene fatto si rischiano problemi di integrità e sicurezza.

```
def validate_json(obj_schema, request_data) -> tuple[bool, Any]:
    # Resto del codice.
```

Se si riscontrano problemi a questo livello l'app ritorna un valore HTTP di `400` (bad request) invece che il classico stato di successo `201`.

In sostanza, questa funzione si occupa di chiamare altre funzioni e classi di validazione spiegate nei prossimi punti.

2.3.3.3 Lettura dati dal database

Per leggere oggetti dal database, che sia un insieme di istanze oppure un singolo oggetto, è sufficiente chiamare direttamente i metodi ORM di SQLAlchemy. Le funzioni `list_db_objects` e `detail_db_object` si occupano di fare le query al database e formattare una risposta JSON. In questo modo il codice necessario negli endpoint definiti in `app.py` viene ridotto, come spiegato nel paragrafo 2.3.2.1.4.1.

```
def list_db_objects(obj_class: Any, db: Any) -> Any:
    try:
        objs = db.session.query(obj_class).all()
        return jsonify([obj.serialize() for obj in objs])
    except Exception as e:
        return jsonify({"error": "could not retrieve objects"}), 500
```

Creiamo una lista di oggetti serializzati dal database. Se sorge qualunque problema, per semplicità, ritorniamo un errore interno al server (HTTP `500`).

Nel caso della vista a dettaglio usiamo quest'istruzione, quindi si filtra semplicemente per id:

```
return db.session.execute(
    select(obj_class).where(obj_class.id == obj_id)
).scalars().first()
```

2.3.4 json_http_schema.py

Questo file definisce le strutture dati per la validazione del JSON originato dalle richieste via gli endpoint HTTP. Questa parte di codice è strettamente legata alla sezione 2.3.3.2.

Senza questi meccanismi qualunque richiesta non conforme verrebbe processata, il che potrebbe portare a degli errori nel backend, oltre che a problemi di sicurezza.

Nel prossimo punto vedremo più nel dettaglio queste classi.

2.3.4.1 Classi

Per effettuare la validazione usiamo una libreria Python chiamata mashumaro[13]. Questa è in grado di *deserializzare*, cioè trasformare un oggetto JSON (quindi una stringa) in una dataclass, già citate nella sezione 2.3.1.3.

Mashumaro è fra i software più veloci nella sua categoria, come spiegato nella pagina del progetto[14], ed

[13] Fast and well tested serialization library: https://github.com/Fatal1ty/mashumaro
[14] Mashumaro - Benchmark: https://github.com/Fatal1ty/mashumaro?tab=readme-ov-file#benchmark

è molto facile da usare, come vedremo in seguito.

Al momento della trasformazione, mashumaro fa implicitamente anche una validazione dei campi e dei tipi di dati.

2.3.4.1.1 Casi base

Prendiamo la classe per la deserializzazione di *Plant*:

```
from mashumaro.mixins.json import DataClassJSONMixin
from dataclasses import dataclass

@dataclass
class PlantSchema(DataClassJSONMixin):
    name: str
    owner_id: int
    plant_module_system_id: int
    plant_battery_system_id: int
    plant_production_id: int
    installer: str
```

In questo caso il codice Python che viene eseguito può essere semplicemente questo:

```
plant_instance = PlantSchema.from_json(json_string)
```

In realtà, viene chiamata la funzione `validate_json` che abbiamo visto nella sezione 2.3.3.2. Così la deserializzazione funziona per tutti gli schemi (dataclass) definiti all'interno di `json_http_schema.py`.

2.3.4.1.2 Casi complessi

In casi più complessi utilizziamo una validazione più avanzata. Per esempio guardiamo lo schema delle specifiche di batterie (*BatterySpecification*):

```
@dataclass
class BatterySpecificationSchema(DataClassJSONMixin):
    type: str
    voltage_range_id: int
    current_range_id: int
    capacity: float

    def __post_init__(self):
        # Prendi uno dei valori validi dalle enumerazioni.
        valid_values: list[str] = [e.value for e in classes_orm.BatteryType]
        if self.type not in valid_values:
            raise ValueError(f'field "name" must be one of {valid_values}')

        self.type = classes_orm.BatteryType[self.type.upper()].value
```

Oltre ai campi abbiamo anche il metodo `__post_init__`. Come si può capire dal nome, questo metodo viene eseguito dopo che l'instanza della classe viene creata.

Nel file dello schema del database sono stati definite alcune enumerazioni Python, cioè una raccolta di chiavi-valori assegnati all'interno di una classe. Nel caso in esame, il nome di una batteria può essere solo `master` o `slave`.

```
class BatteryType(Enum):
    MASTER = 'master'
    SLAVE = 'slave'
```

Se, nel momento della deserializzazione il campo `type` non corrisponde ad uno dei due tipi, allora viene ritornato un errore che poi sarà inviato come risposta HTTP JSON.

Altro esempio, ma questa volta con ereditarietà: qui andiamo a controllare se il valore più basso è davvero minore uguale a quello più alto.

```
@dataclass
class LimitSchema(DataClassJSONMixin):
    lower: float
    upper: float

    # upper must be > lower.
    def __post_init__(self):
        if self.lower >= self.upper:
            raise ValueError('field "lower" must be greater than field "upper"')
```

Per risparmiare codice, le classi dei range vanno ad ereditare tutte le informazioni dalla classe base `LimitSchema`. Qui riportate solo due classi:

```
@dataclass
class VoltageRangeSchema(LimitSchema):
    pass

@dataclass
class CurrentRangeSchema(LimitSchema):
    pass
```

Capitolo 3

Frontend

Il frontend è uno dei due modi, insieme all'API, con cui l'utente finale può interagire con l'app.

In questo progetto i tecnici che monitorano gli impianti di produzione di energia utilizzeranno un normale web browser per collegarsi all'app, e non è necessario che siano a conoscenza delle API. Solo chi sviluppa sistemi di sensori collegati a microcontrollori deve fare in modo che questi possano inviare i dati attraverso HTTP all'API discussa nella parte di backend.

3.1 Struttura

Tutto il codice di questa sezione si trova all'interno della directory `frontend`. Si è inizializzato un nuovo progetto Svelte[1] usando il software Vite[2] che è predisposto con alcuni template.

I progetti Svelte sono organizzati in componenti. Per rispecchiare la struttura del database esiste un componente software scritto in Svelte per ogni modello. Questi componenti sono orientati alla visualizzazione dei dettagli degli oggetti. Altri componenti hanno invece il compito di visualizzare grafici oppure provvedono all'inserimento o alla modifica dei dati.

Così come fatto per il backend, anche qui andremo a vedere le varie parti, le motivazioni delle scelte di questi, e le loro interazioni.

3.2 Strumenti utilizzati

3.2.1 Svelte

Ora vediamo più nel dettaglio che cosa sono Svelte e Vite.

Svelte è un framework JavaScript usato per creare interfacce utente, che agisce anche da compilatore a JavaScript. Rispetto ad altri framework, come React o Vue.js, Svelte si caratterizza per avere una curva di apprendimento più facile, essere provvisto di meno componenti esterne, e soprattutto avere un sistema di *reattività* integrato.

Come accennato prima, un progetto di questo tipo viene diviso in componenti. Ogni componente è contenuta in un file Svelte. La struttura tipica di un componente è questa:

[1] Svelte • Web development for the rest of us: https://svelte.dev/
[2] Vite | Next Generation Frontend Tooling: https://vite.dev/

```
<script>
// Codice
</script>
<!--HTML-->
<style>
/* CSS */
</style>
```

La prima sezione è quella del codice Svelte e JavaScript. Qui vengono posizionate le funzioni e le variabili che devono essere usate nella sezione dell'HTML. Oltre ai classici idiomi di JavaScript si possono usare anche le istruzioni specifiche di Svelte, come per esempio la *reattività*.

La parte centrale consiste nel classico HTML mescolato a Svelte. Le variabili e istruzioni Svelte sono racchiuse all'interno di parentesi graffe. In questo modo si possono usare anche costrutti quali i cicli, le condizioni, etc. Per certi aspetti il funzionamento è simile ai template Jinja[3], usati da Flask, o ai layout di Jekyll[4].

Nel progetto, la sezione di stile dei componenti non è molto utilizzata. La maggior parte degli stili è infatti definita nel file app.css, e quindi i componenti vanno ad utilizzare quelle classi CSS.

3.2.2 Vite

Vite è uno strumento che permette la creazione di template di progetti per framework JavaScript. Ha incluso anche comandi per fare il build e per vedere il risutato delle modifiche al codice in tempo reale.

Una volta installato il gestore di pacchetti Javascript npm[5], lo standard di fatto per JavaScript come può essere pip per Python, si può creare un nuovo progetto Svelte in questo modo:

```
npm create vite@latest frontend -- --template svelte
```

Una volta fatte le modifiche possiamo fare il build:

```
cd frontend
# Modifiche...
npm run build
```

Durante lo sviluppo ed anche durante il deployment non si è dovuto usare il server integrato di Vite. Infatti, avendo già il server Flask in attività, e necessitando di comunicare con l'API, tutte le chiamate al backend fatte con un browser collegato al server di sviluppo Vite non funzionano: ci sono due porte di ascolto, Flask e npm, quindi il browser utilizza solo quella di npm. La soluzione sta semplicemente nel compilare il codice in modalità produzione e collegarsi al server Flask.

3.2.3 Chart.js

Chart.js[6] è una libreria JavaScript che permette di creare grafici in modo semplice. Rispetto ad altre librerie, come per esempio D3.JS[7], è più semplice da usare anche se ha alcune limitazioni nella personalizzazione. In questo progetto, comunque, abbiamo bisogno solo di alcuni grafici di base, come quelli a barre e a linee.

Il procedimento per creare un grafico è il seguente:

[3] Jinja is a fast, expressive, extensible templating engine: https://jinja.palletsprojects.com/en/stable/
[4] Layouts | Jekyll • Simple, blog-aware, static sites: https://jekyllrb.com/docs/step-by-step/04-layouts/
[5] npm è un gestore di pacchetti per il linguaggio di programmazione JavaScript: https://www.npmjs.com/
[6] Simple yet flexible JavaScript charting library for the modern web: https://www.chartjs.org/
[7] The JavaScript library for bespoke data visualization: https://d3js.org/

1. ottenimento dei dati dall'API
2. formattazione dei dati. Chart.js si aspetta un oggetto avente etichette e dati, come in questo esempio:

```
const data = {
  labels: timestamps
  datasets: [
    {
       label: 'AC Voltage',
       data: sensorReadings.AC.voltage,
       borderColor: 'rgba(75, 192, 192, 1)',
       fill: false,
    },
    {
       label: 'AC Current',
       data: sensorReadings.AC.current,
       borderColor: 'rgba(75, 192, 192, 1)',
       fill: false,
    },
    // ecc...
  ]
}
```

3. aggiunta opzioni al grafico come tipo di grafico, colori, etichette, ecc...
4. plot

3.3 Visualizzazione dati

In questa sezione vedremo come si presenta l'interfaccia web.

Quando l'utente si collega alla dashboard gli viene presentato un prospetto di tutti gli impianti e di alcune operazioni che si possono compiere.

Figura 3.1: La home

Da qui ci si può spostare su uno specifico impianto cliccando sul nome:

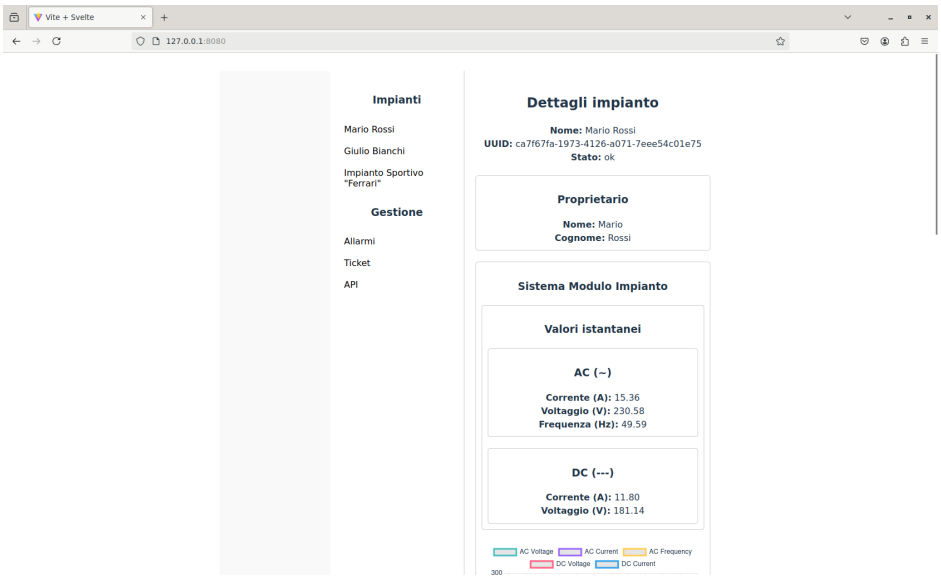

Figura 3.2: Esempio di vista impianto

L'*entry point*, cioè il punto di partenza del frontend di questa app è il componente `PlantDetails.svelte` e, come si può intuire dal nome, rappresenta la vista a dettaglio di un impianto. Ecco com'è definito all'interno di `main.js`:

```
import { mount } from 'svelte'
import './app.css'
import App from './show/PlantDetails.svelte'

const app = mount(App, {
  target: document.getElementById('app'),
})

export default app
```

Da notare che qui importiamo anche il file contentente i CSS più comunemente usati.

3.3.1 Significato dei dati

La vista di ogni impianto è divisa in quattro parti:

- dati base
- sistema modulo impianto
- sistema batterie impianto
- dati produzione

Adesso vedremo come si presentano a livello visivo di web browser ed anche a livello di componente Svelte.

3.3.1.1 Dati base

Per identificare un impianto di può usare il campo UUID oppure il nome.

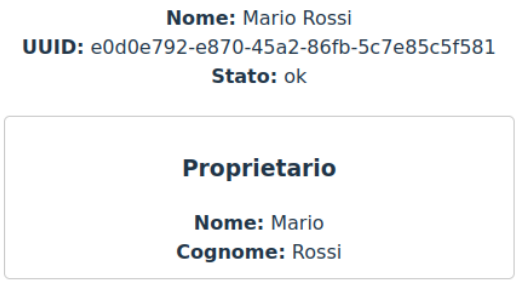

Figura 3.3: Dati base impianto

A livello di componente Svelte, in particolare per questi dettagli, possiamo vedere come è semplice recuperare e visualizzare i dati. Per esigenze di lettura viene mostrata solo una piccola parte.

```
<h2>Dettagli impianto</h2>
<ul>
    <li><strong>Nome:</strong> {$plantDetails.name}</li>
    <li><strong>UUID:</strong> {$plantDetails.uuid}</li>
    <li><strong>Stato:</strong> {$plantDetails.status}</li>

    <li>
        {#if $plantDetails.owner_id}
            <OwnerDetails ownerId={$plantDetails.owner_id} />
        {/if}
    </li>

    <!--Altro codice.-->

</ul>
```

La parte iniziale dell'HTML è intuitiva: una volta che abbiamo recuperato i dettagli li andiamo a visualizzare attributo per attributo.

Per quanto riguarda le informazioni dei proprietari dell'impianto, andiamo a importare il componente *OwnerDetails* che si trova nel file `OwnerDetails.svelte`, passando l'id del proprietario ricavato dall'istanza dell'impianto.

```
<script>
import { writable } from 'svelte/store';
import { fetchResourceDetails } from '../utils/showUtils.js';
import OwnerDetails from './OwnerDetails.svelte';

const plantDetails = writable(null);

async function fetchPlantDetails(plantId) {
    const data = await fetchResourceDetails('/plant', `${plantId}`);
    plantDetails.set(data);
}
</script>
```

Nella sezione JavaScript abbiamo la funzione `fetchPlantDetails`. Questa va a recuperare i dati dall'API e a settare uno *store* Svelte. Uno store in questo contesto è un particolare tipo di oggetto che è anche reattivo, cioè il valore salvato può cambiare automaticamente se degli eventi, spesso provocati dall'utente, vanno a interagire con altri componenti legati ad esso. In questo esempio, quando andiamo

a selezionare un'altro impianto dalla lista, tutti i dettagli dell'impianto fino a quel momento visualizzato vengono sostituiti:

```
<script>
const plantList = writable([]);
const selectedPlantId = writable(null);

async function fetchPlants() {
    const data = await fetchResource('/plant');
    plantList.set(data);
}

function selectPlant(plantId) {
    /* Resetta la vista corrente: fai in modo che gli impianti
     * possano essere selezionati. */
    currentView.set(null);

    selectedPlantId.set(plantId);
    fetchPlantDetails(plantId);
}
</script>
```

Per cambiare impianto utilizziamo l'elemento *button* dell'HTML, ed usiamo l'evento *on:click* che chiama la funzione `selectPlant`.

La lista degli impianti è ottenuta con un'altra chiamata API sull'endpoint `/plant`.

```
<h3>Impianti</h3>
{#if $plantList.length > 0}
    {#each $plantList as plant}
        <button class="plant-item" on:click={() => selectPlant(plant.id)}>
            {plant.name}
        </button>
    {/each}
{:else}
    <p>Loading plants...</p>
{/if}
```

3.3.1.2 Recupero risorse

I dati dall'API vengono recuperati con le funzioni JavaScript `fetchResourceDetails` e `fetchResource`. Queste si occupano di fare una chiamata HTTP `GET` sulla risorsa specificata, e di trasformare la risposta in un oggetto JSON:

```
export async function fetchResourceDetails(endpoint, id) {
    try {
        const response = await fetch(`${endpoint}/${id}`);
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error fetching resource:', error);
        throw error;
    }
}
```

3.3.1.3 Grafico e dati sistema modulo impianto

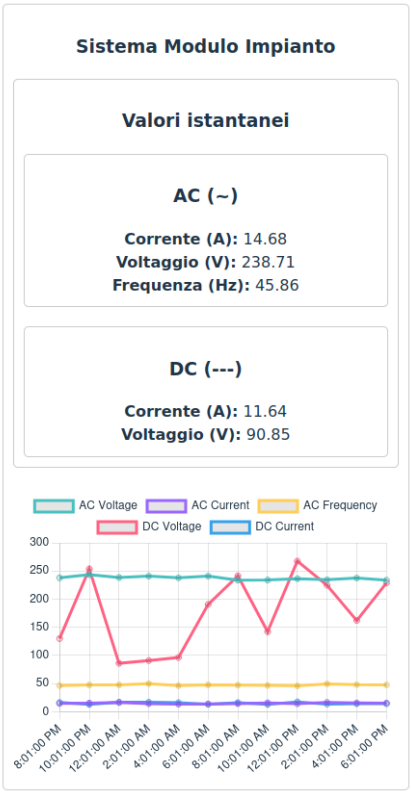

Figura 3.4: Sistema modulo impianto

3.3.1.3.1 Valori numerici Tornando alla vista dal browser, i valori numerici, come indicato dall'intestazione, rappresentano gli ultimi dati noti, quindi istantanei, dell'impianto. Il primo grafico, invece, rappresenta un riepilogo degli stessi dati nel corso delle ultime 24 ore. La larghezza di ogni intervallo è di due ore, quindi è stato necessario calcolare una media aritmetica dei valori tra un intervallo e l'altro.

Per visualizzare tutti questi dati abbiamo, ancora una volta, dei sotto-componenti chiamati al bisogno. Questi sono raccolti all'interno di *PlantModuleSystemDetails.svelte*:

```
<div class="object-details">
    <h3>Sistema Modulo Impianto</h3>
    {#if $plantModuleSystem}
        {#if $plantModuleSystem.plant_id}
            <PlantModuleSystemSensorReadingsDetails plantId={$plantModuleSystem.plant_id} />
        {/if}
        <br />
        <PlantModuleSystemSensorReadingsChart plantId={$plantModuleSystem.plant_id} />
    {:else}
        <p>Caricamento dettagli sistema modulo impianto...</p>
    {/if}
</div>
```

I dati delle letture vengono visualizzati in questo modo:

```
<li><b>Corrente (A):</b> {ac_current.toFixed(2)}</li>
<li><b>Voltaggio (V):</b> {ac_voltage.toFixed(2)}</li>
<li><b>Frequenza (Hz):</b> {ac_frequency.toFixed(2)}</li>
```

Con il metodo `toFixed()` manteniamo solo due cifre decimali in modo da rendere la lettura più semplice.

I dati vengono recuperati dall'endpoint, conoscendo il numero dell'impianto `plantId` passato dal componente chiamante:

```
const response = await
↪  fetch(`/plant_module_system_sensor_reading?plant_id=${plantId}&results=25`);
```

Come detto, in realtà, ci interessa solo l'ultimo dato di ogni elemento, che sappiamo essere il primo della lista perchè è ordinata per timestamp decrescente all'interno della risposta JSON:

```
<script>
let dc_voltage = 0, dc_current = 0, ac_voltage = 0, ac_current = 0, ac_frequency = 0;

// Altro codice.

let ac = false, dc = false;
for (const reading of data) {
    const { voltage, current, frequency } = reading;
    if (frequency === 0.0) {
        dc_voltage = voltage;
        dc_current = current;
        dc = true;
    }
    else {
        ac_voltage = voltage;
        ac_current = current;
        ac_frequency = frequency;
        ac = true;
    }
    if (ac && dc)
        break;
}
</script>
```

3.3.1.3.2 Grafico Ora abbiamo l'opportunità di analizzare il codice che disegna il grafico con Chart.js. Anche in questo caso il componente chiamante *PlantModuleSystemDetails* passa l'ID dell'impianto. Gran parte del codice viene omesso ma vi è l'aggiunta di alcuni commenti.

```
<script>
import { Chart, registerables } from 'chart.js';

export let plantId;

function createChart() {
    // Contesto, cioè un ambiente dove poter disegnare il grafico.
    // Vedere la sezione HTML sotto.
    const ctx = document.getElementById('sensorReadingsChart').getContext('2d');

    // Dati preparati come spiegato precedentemente.
    const data = { ... }
```

```
    // Elimina un grafico precedente se esiste.
    if (chart) {
      chart.destroy();
    }

    // Qualche opzione extra.
    let baseOptions = { ... }

    // Oggetto aggregato per le opzioni da usare poi nel grafico.
    // manageChartColorScheme() serve per visualizzare il grafico quando è
    // attivo il tema scuro del browser.
    const chartOptions = {...baseOptions, ...manageChartColorScheme()};

    // Il plot vero e proprio.
    chart = new Chart(ctx, {
      type: 'line', // Line chart
      data: data,
      options: chartOptions,
    })
}
</script>

/* Da notare l'id `sensorReadingsChart` che deve essere uguale
 *a quello del contesto. */
<canvas id="sensorReadingsChart" width="400" height="300"></canvas>
```

Per il calcolo delle medie esiste una funzione specifica, `processSensorReadings`:

```
<script>
let sensorReadings;

function processSensorReadings(data) {
    data.forEach((reading) => {
        // Estraiamo i valori dall'oggetto lettura.
        const { voltage, current, frequency, timestamp } = reading;

        // Calcoliamo l'età in ore della lettura
        const hoursAgo = ...

        /* Selezionamento l'intervallo di riferimento, cioè dove posizionare
         * la lettura. Dobbiamo mettere i vari valori della lettura in uno dei
         * bucket. */
        const intervalIndex = ...

        /* Aggiungiamo i valori ai bucket e teniamo traccia di quanti elementi
         * ci sono in ogni bucket. Qui vediamo solo la corrente continua (DC). */
        sensorReadings.DC.voltage[intervalIndex] += voltage;
        sensorReadings.DC.current[intervalIndex] += current;
        sensorReadings.DC.counts[intervalIndex] += 1;

        // Calcolo media.
        for (let i = 0; i < chart_x_intervals; i++)
        {
            sensorReadings.DC.voltage[i] /= sensorReadings.DC.counts[i];
            sensorReadings.DC.current[i] /= sensorReadings.DC.counts[i];
        }
```

```
    }
}
</script>
```

3.3.1.4 Grafico e dati sistema batteria impianto

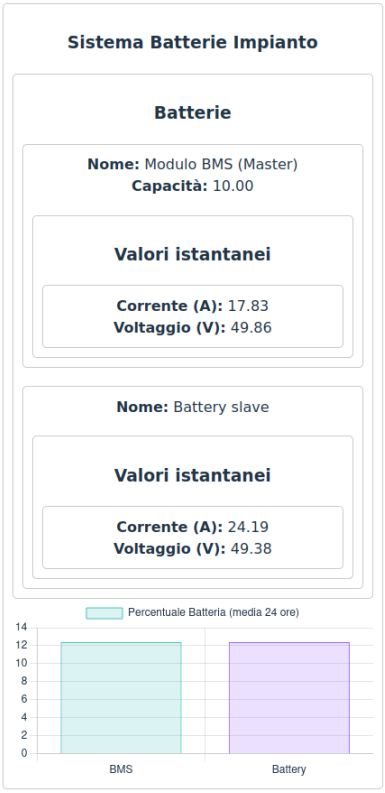

Figura 3.5: Sistema batteria impianto

I dati delle due batterie funzionano in modo simile ai precedenti. Vengono presentati i dati istantanei e la percentuale media di carica di entrambe le batterie nelle ultime 24 ore. Il codice è simile e in generale più semplice di quello visto per il sistema modulo impianto.

3.3.1.5 Grafico e dati produzione impianto

I dati di produzione impianto invece sono tutti istantanei. Il tipo di grafico è a barre orizzontale.

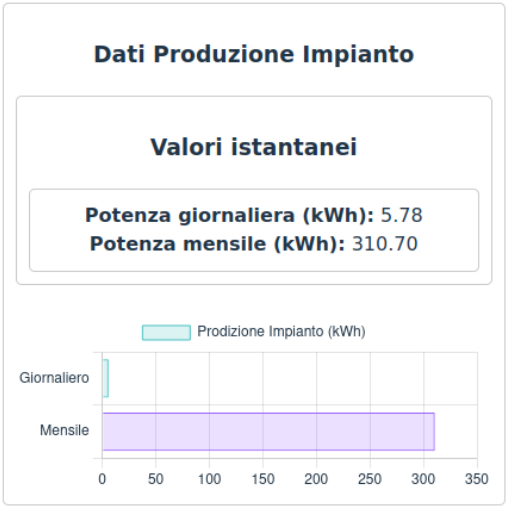

Figura 3.6: Dati produzione impianto

Per farlo di questo tipo è sufficiente specificare l'asse che funge da indice, che in questo caso è l'asse y:

```
let baseOptions = {
        indexAxis: 'y',
        ...
}
```

3.3.2 Allarmi

Nella web UI, nella parte a sinistra esiste la voce *Allarmi*. Abbiamo già discusso quest'argomento nelle sezioni 2.3.1.1.7 e 2.3.2.1.6, rispettivamente nello schema del database e nell'API REST. Ora vediamo come funziona la parte grafica.

Allarmi

Nome impianto	UUID impianto	Timestamp (UTC)	Descrizione	Codice	Gravità
Giulio Bianchi	b94f4c7b-d3fc-4fd4-8f5f-d11ba399ebde	2025-02-14 17:54:34	problema batteria slave	03	medio
Mario Rossi	52594d41-6076-4dd8-88fe-75203a630cf5	2025-02-14 17:54:22	errore software	00	medio
Giulio Bianchi	b94f4c7b-d3fc-4fd4-8f5f-d11ba399ebde	2025-02-14 17:54:22	problema componenti impianto	04	medio
Mario Rossi	52594d41-6076-4dd8-88fe-75203a630cf5	2025-02-14 17:54:10	problema componenti impianto	04	medio
Impianto Sportivo "Ferrari"	937af01f-c761-4b66-8dd0-fe98e34452d9	2025-02-14 17:53:58	problema componenti impianto	04	medio
Impianto Sportivo "Ferrari"	937af01f-c761-4b66-8dd0-fe98e34452d9	2025-02-14 17:53:58	problema batteria slave	03	medio
Mario Rossi	52594d41-6076-4dd8-88fe-75203a630cf5	2025-02-14 17:53:46	problema CC inverter	01	medio
Impianto Sportivo "Ferrari"	937af01f-c761-4b66-8dd0-fe98e34452d9	2025-02-14 17:53:46	problema componenti impianto	04	medio

Figura 3.7: Allarmi

In questo caso gli allarmi sono visualizzati e ordinati in ordine decrescente di timestamp, così i più recenti si trovano all'inizio della tabella. Qui l'utente può solo visualizzare e non interagire.

3.3.3 Ticket

I ticket, come già detto precedentemente, sono collegati agli allarmi. La tabella è simile a quella degli allarmi, ma qui si può anche gestire lo stato.

Quando il sistema crea automaticamente un nuovo ticket, questo ha `NOT RESOLVED` come valore di stato. L'utente può, se necessario, impostare uno dei tre stati con il menù a tendina, cioe:

- `NOT RESOLVED`
- `IN PROGRESS`
- `RESOLVED`

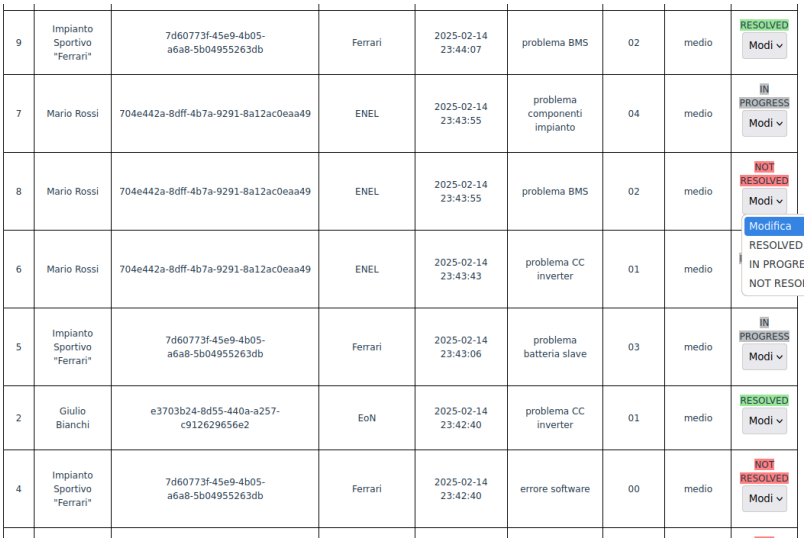

Figura 3.8: Ticket

Per la gestione dello stato è stata implementata l'unica operazione di `PUT` di questo progetto. Si trova nel componente *TicketDetails.svelte*. Il menù a tendina chiama la funzione `updateTicketStatus` con il valore selezionato, `e.target.value`:

```
<select on:change={(e) => updateTicketStatus(ticket.id, e.target.value, e)}>
    <option value="">Modifica</option>
    <option value="RESOLVED">RESOLVED</option>
    <option value="IN PROGRESS">IN PROGRESS</option>
    <option value="NOT RESOLVED">NOT RESOLVED</option>
</select>
```

L'operazione di aggiornamento deve agire solo su uno specifico ticket, per cui si deve passare l'ID.

```
<script>
async function updateTicketStatus(ticketId, selectedStatus, event) {
    /* Controlla se è stato passato uno stato valido. Se è `undefined`
     * non ignora l'operazione. */
    if (!selectedStatus)
        return;

    const response = await fetch(`/ticket/${ticketId}`, {
        method: 'PUT',
        headers: {
```

```
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ status: selectedStatus }),
    });

    /* Aggiorna la lista dei ticket in modo che il ticket modificato
     * rifletta il nuovo stato. */
    await fetchTickets();
}
</script>
```

Il ticket prende le informazioni rilevanti dall'allarme più recente in termini di timestamp associatogli. Per questo motivo si vede spesso la notazione `ticket.alarms[0]` all'interno del componente Svelte. Sappiamo infatti che gli allarmi sono già ordinati in ordine decrescente di timestamp.

3.3.4 API

La sezione di API nel frontend mostra un rieilogo base di alcuni dati ed è principalmente pensato per gli sviluppatori software. C'è infatti una colonna chiamata `Token ID`. Questo valore viene usato come metodo di autenticazione per fare alcune operazioni con l'API REST e deve essere parte del *payload* JSON.

API

Nome impianto	UUID impianto	Marca inverter	Marca batterie	Token ID
Mario Rossi	9820b705-92ef-4719-8c93-ed82665cffcd	Agip	Repsol	hello-0-a
Giulio Bianchi	b71e0d0a-8b0a-47c4-947e-8956e59ae0d2	EoN	Agip	hello-0-a
Impianto Sportivo "Ferrari"	fe9cf35d-2d01-471b-ba59-42a307101330	EoN	Repsol	hello-0-a

Figura 3.9: API

3.4 Creazione impianto

3.4.1 Grafica

I nuovi impianti possono essere creati dalla dashboard cliccando sul bottone che si trova sulla sinistra.

Figura 3.10: Bottone per la creazione di un impianto

Figura 3.11: Mashera creazione impianto

Come si vede dalle immagini si tratta solo di una semplice dimostrazione. L'implementazione è incompleta ma rende comunque l'idea di quello che si può fare. Per esempio si può creare un nuovo proprietario direttamente da questa maschera. Associare un proprietario ad un nuovo impianto è infatti uno dei requisiti per la creazione di quest'ultimo.

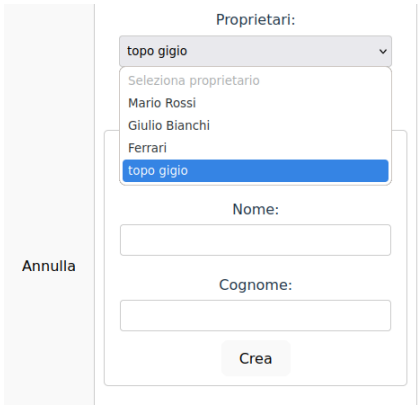

Figura 3.12: Maschera e lista proprietari

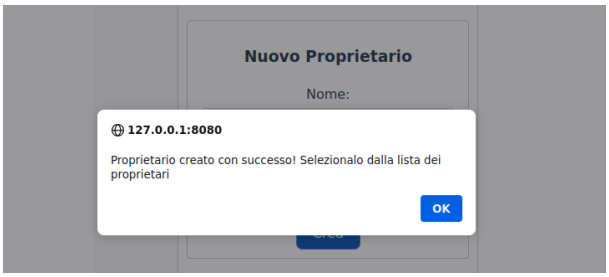

Figura 3.13: Messaggio operativo

3.4.1.1 Codice

Il codice Svelte per fare queste cose è diviso in due componenti: *NewOwner.svelte* e *NewPlant.svelte*. Il primo è il componente che crea i nuovi proprietari, ed è importato dal secondo che crea gli impianti. Questi codici ci danno modo di vedere come funziona l'invio dei dati (`POST`).

Partiamo dalla creazione dei proprietari con il form. Qui vediamo solo il campo `first_name`:

```
<form on:submit={handleSubmit}>
    <div>
        <label for="first_name">Nome:</label>
        <input type="text" id="first_name" bind:value={$newOwner.first_name} required />
    </div>
    <button type="submit">Crea</button>
</form>
```

Quindi, quando l'utente clicca su `Crea`, viene lanciata la funzione `handleSubmit` e viene scritto il valore nello *store* `newOwner`:

```
<script>
import { createEventDispatcher } from 'svelte';
import { submit } from '../utils/formUtils.js';

const newOwner = writable({
    first_name: '',
    last_name: '',
});

// Definiamo un evento.
const dispatch = createEventDispatcher()

async function handleSubmit(event) {
    const result = await submit(event, '/owner', $newOwner);
    alert('Proprietario creato con successo! Selezionalo dalla lista dei proprietari');

    /* Molto importante: l'evento di creazione viene passato dal componente
     * figlio (NewOwner) al Padre (NewPlant). */
    dispatch('ownerCreated');

    // Reset dei dati dall'oggetto.
    newOwner.set({ first_name: '', last_name: '' })
}
</script>
```

La funzione `submit` si occupa di mandare i dati con usa richiesta di tipo `POST`, allo stesso modo di

quelle viste per il terminale con *curl*:

```
export async function submit(event, endpoint, data) {
    const response = await fetch(endpoint, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(data),
    });
    return await response.json();
}
```

Tornando al componente principale, *NewPlant.svelte*, anche qui abbiamo lo stesso meccanismo con il form. Questo ha i vari menù a tendina, compreso quello dei proprietari. Gli altri campi, però, funzionano solo con gli ID del

database. Il menù con la lista dei proprietari è implementato in modo molto semplice con un ciclo `for`.

Può essere anche interessante sapere come comunica il componente padre *NewPlant,velte* con il figlio *NewOwner.svelte*:

```
<label for="owner">Proprietari:</label>
<select id="owner" bind:value={$newPlant.owner_id} required>
    <option value="" disabled>Seleziona proprietario</option>
    {#each $owners as owner}
        <option value={owner.id}>{owner.first_name} {owner.last_name}</option>
    {/each}
</select>

<!--Importiamo il componente e definiamo un evento che verrà eseguito alla
    creazione di un proprietario.-->
<NewOwner on:ownerCreated={handleOwnerCreated} />
```

La seguente funzione di aggiornamento dei proprietari verrà chiamata solo una volta che la creazione del proprietario è completata, specificamente con la funzione `dispatch('ownerCreated')`, che "invia" l'evento. Il componente padre è quindi in "ascolto" di questo evento.

```
<script>
function handleOwnerCreated() { fetchOwners(); }
</script>
```

Capitolo 4

Script di popolamento

Gli script di popolamento hanno lo scopo di fornire una serie di dati di esempio plausibili per dimostrare l'utilizzo dell'app. Per questione pratiche si è deciso di usare un database *stateless*, che cioè non lavora su file ma totalmente in memoria RAM. In un'applicazione usata a livello di produzione si utilizzerebbe un database tradizionale, almeno di tipo SQLite, in modo che anche al riavvio i dati siano persistenti. Avere un database sempre vuoto ci consente però di popolarlo di dati in modo deterministico ed evitare ulteriori controlli. È qui che entrano in gioco gli script.

4.1 Struttura

Questi programmi sono molto semplici e si occupano di interagire con l'API REST della nostra app. Ogni script è separato dall'altro all'interno di directory divise per endpoint. L'utilizzo di questi programmi è stato utile anche durante lo sviluppo perchè hanno permesso di testare l'API:

- 00_voltage_range
- 01_current_range
- 02_frequency_range
- 03_daily_power_range
- 04_monthly_power_range
- 05_owner
- 06_token
- 10_battery_specification
- 11_battery
- 12_dc_current_system
- 13_ac_current_system
- 20_plant_module_system
- 21_plant_battery_system
- 22_plant_production
- 30_plant
- 40_plant_module_system_sensor_reading
- 41_plant_battery_system_sensor_reading
- error_populate.py
- list.py
- populate.py

L'utilizzo di ID numerici prima del nome delle tabelle è un aspetto fondamentale. Bisogna infatti ricordarsi che il database deve essere popolato in ordine a causa delle relazioni tra tabelle.

Un altro script, chiamato `populate.py`, va ad eseguire ricorsivamente tutti i file che hanno come parte iniziale del nome `new_`, che, come suggerisce il nome, vanno ad aggiungere dati al database. Ognuna di queste directory tipicamente contiene tre file. Vediamo l'esempio di `00_voltage_range`:

- list_voltage_range.py
- list_voltage_ranges.py
- new_voltage_ranges.py

4.2 Vista dati

Esistono quindi anche script per visualizzare gli elementi. Tipicamente, per visualizzare un singolo elemento si utilizza una struttura come la seguente. L'esempio riportato è per visualizzare il primo range di corrente:

```
#!/usr/bin/env python3

import httpx
import logging
import sys
import pprint

URL: str = 'http://127.0.0.1:8080/current_range/1'
TOKEN: str = 'dummy'

logging.basicConfig(level=logging.INFO)

response = httpx.get(
    URL,
    headers={
        'Authorization': f'Bearer {TOKEN}',
    }
)
pprint.pprint(response.json())
```

Con la libreria httpx[1] l'interazione è molto semplice.

Per visualizzare tutti gli elementi di un endpoint è sufficiente eliminare l'ID alla fine della query. Lo script `list.py` funziona in modo analogo a `populate.py` per quanto riguarda le letture di elementi.

4.3 Inserimento

Per lo script di inserimento, `new_voltage_ranges.py` in questo esempio, il codice deve essere commentato e formattato a dovere per evitare di inserire erroneamente dati incongruenti. Per semplicità si definisce una variabile oggetto che diventa poi il payload JSON nella richiesta HTTP `POST`.

```
VOLTAGE_RANGES: list[dict] = [
    ## PlantModuleSystem ##
    # Mario Rossi - DC.
    # 1
    {'lower': 47.0, 'upper': 319.0},

    # Giulio Bianchi - DC.
    # 2
```

[1] HTTPX is a fully featured HTTP client for Python 3, which provides sync and async APIs, and support for both HTTP/1.1 and HTTP/2: https://www.python-httpx.org/

```
    {'lower': 47.0, 'upper': 319.0},

    # Impianto Sportivo "Ferrari" - DC.
    # 3
    {'lower': 300.0, 'upper': 600.0},

    # Mario Rossi - AC.
    # 4
    {'lower': 230.0, 'upper': 245.0},

    # Giulio Bianchi - AC.
    # 5
    {'lower': 230.0, 'upper': 245.0},

    # Impianto Sportivo "Ferrari" - AC.
    # 6
    {'lower': 200.0, 'upper': 400.0},

    ## PlantBatterySystem ##
    # Mario Rossi - Master.
    # 7
    {'lower': 49.0, 'upper': 50.0},

    # Mario Rossi - Slave.
    # 8
    {'lower': 49.0, 'upper': 50.0},

    # Giulio Bianchi - Master.
    # 9
    {'lower': 49.0, 'upper': 50.0},

    # Giulio Bianchi - Slave.
    # 10
    {'lower': 49.0, 'upper': 50.0},

    # Impianto Sportivo "Ferrari" - Master.
    # 11
    {'lower': 160.0, 'upper': 600.0},

    # Impianto Sportivo "Ferrari" - Slave.
    # 12
    {'lower': 160.0, 'upper': 600.0},
]
```

Esistono anche script più complicati come `new_plant_module_system_sensor_readings.py`. In questo caso le letture dei sensori sono dati casuali ma all'interno di range ben precisi. Esiste anche un sistema di probabilità pesata per ottenere più o meno codici di allarme nelle letture in base ad una percentuale. Di default il 25% delle letture conterrà un codice di allarme, che quindi sarà diverso da `-01` (non allarme). Segue l'estratto del codice commentato:

```
import random

no_alarm_percent_probability: float = 75.0
```

Lista di possibili codici allarme:

```
alarm_codes: list[str] = ['-01', '00', '01', '02', '03', '04']
```

La probabilità deve essere normalizzata a 1. La percentuale è solo un modo più intuitivo per rappresentarla.

```
normalized_no_alarm_probability: float = no_alarm_percent_probability / 100
```

Bisogna calcolare la probabilità di allarme (`!= -01`), che è l'inverso della probabilità di non allarme (`== -01`). La probabilità si riferisce ad ogni singolo codice di allarme, per cui la lista delle probabilità sarà: $[p_a, p_n, p_n, p_n, p_n, p_n]$, con p_a come probabilità di allarme, mentre p_n come probabilità di non allarme. Questa è anche nota come lista dei pesi. Il calcolo di p_n è il seguente:

```
alarm_probability = (1 - normalized_no_alarm_probability) / (len(alarm_codes) - 1)
```

Creiamo quindi la lista dei pesi: la somma di tutti questi elementi dà sempre 1 (100%). La probabilità di tutti i codici che risultano in un allarme, quindi non `-01`, è uguale per tutti i codici. La probabilità per `-01` è invece indicata da `no_alarm_percent_probability`. Il metodo `random.choices` fa la scelta.

```
remaining_slots: int = len(alarm_codes) - 1
weights: list[int] = [normalized_no_alarm_probability] + [alarm_probability] *
↪  remaining_slots
alarm_code = random.choices(alarm_codes, weights, k=1)[0]
```

4.4 Letture sensori continue e simulazione di errori

Sempre rimamendo sul tema delle letture di sensori esistono anche altri script. Per avere letture continue, in modo che i grafici siano più plausibili, un altro programma si occupa di chiamare in continuazione una funzione all'interno di `new_plant_module_system_sensor_readings`.

La app può anche essere lanciata in modo da simulare diversi errori sui componenti, oppure semplicemente problemi causati da errori software procurati dagli sviluppatori. Per questi casi ci sono gli script `error_new_*.py` all'interno di `40_plant_module_system_sensor_reading` e `41_plant_battery_system_sensor_reading`. Questi programmi sono lanciati dallo script `error_populate.py` e vengono usati come override ai due script normali.

Capitolo 5

Deployment

In questo capitolo vedremo come far funzionare l'app e qual è il processo di installazione. In generale, comunque, il deployment è molto simile per i tre principali sistemi operativi: GNU/Linux, MacOS e Windows.

Il progetto è stato testato su Debian 12 e Windows 10, rispettivamente usando Python versione 3.11 e 3.12.

Per prima cosa è necessario installare Python in una delle due versioni indicate ed npm.

Fatto ciò bisogna creare un ambiente Python virtuale con *venv*[1]: questo eviterà di "sporcare" il sistema durante l'installazione dei pacchetti perchè verranno creati in un ambiente isolato. Infine bisogna installare i pacchetti JavaScript con npm.

L'avvio comprende l'app e gli script di popolamento: infatti l'app parte sempre senza dati in quanto il database è esclusivamente in memoria RAM.

5.1 Linux

Su Debian tutto il procedimento risulta semplice per via della presenza di tutti gli strumenti necessari nel gestore di pacchetti. Passiamo all'installazione:

```
sudo apt install python3 python3-venv npm
```

Ora è necessario installare le dipendenze del frontend e compilarlo:

```
cd frontend
npm install
npm run build
```

Creiamo l'ambiente virtuale, installiamo le dipendenze Python ed avviamo il programma:

```
python3 -m venv .venv
. .venv/bin/activate
pip install -r requirements
python app.py
```

Infine, su un altro terminale si possono eseguire gli script di popolamento per ottenere dei dati di esempio:

[1] venv — Creation of virtual environments: https://docs.python.org/3.11/library/venv.html

```
cd scripts
python -m populate
cd 40_plant_module_system_sensor_reading
python -m continuous_new_sensor_readings
```

Come alternativa si può usare lo script `run.sh`. Questo esegue tutti i comandi necessari per l'avvio. Prima dell'utilizo è necessario aver già creato l'ambiente virtuale Python ed avere le dipendenze installate, così come quelle del frontend.

5.2 Windows

L'installazione e l'avvio su Windows richiede qualche accorgimento in più.

npm va installato dal sito ufficiale di Node JS, mentre Python 3.12 si ottiene dallo store ufficiale di Windows.

Per l'avvio di script arbitrari è necessario dare alcuni permessi. Aprendo una finestra Powershell come amministratore si deve lanciare questo comando:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

È importante notare che questo può esporre a problemi di sicurezza e si consiglia di resettare le impostazioni di default una volta finito l'utilizzo della web app.

Tutti gli altri comandi vanno eseguiti in una finestra Powershell normale. L'unico comando che differisce leggermente è quello di attivazione dell'ambiente virtuale:

```
. .\.venv\Scripts\activate
```

5.3 MacOS

Questo sistema è più simile a Linux rispetto a Windows e Fondamentalmente le istruzioni dovrebbero essere molto simili a quelle del sistema libero. Quello che cambia è l'installazione dei pacchetti di sistema che avviene con altri mezzi.

Capitolo 6

Conclusioni

Lo sviluppo di questo programma ha comportato diverse sfide. La più importante è stata capire l'instabilità del database e di Flask quando usati in modalità sviluppo. Non ci sono molte informazioni a riguardo, soprattutto su SQLAlchemy. Un aspetto da poter migliorare potrebbe essere quindi quello di testare il server di produzione di Flask ed un database più robusto come PostgreSQL. Per fare questo bisogna sicuramente implementare nuovi script per la gestione del database e di Flask.

Altro aspetto crtitico è stato lo sviluppo del frontend. vite, lo strumento di build per Svelte, non dà nessuna indicazione quando si verifica un errore di programmazione. Un modo per aver più informazioni è quello di lanciare il build con `npm run dev`, ma, come detto prima, questa modalità non si integra bene con il progetto.

La gestione degli script di popolamento è stata adattata in modo che funzionasse anche per Windows. Normalmente però un'app di questo tipo, a livello di produzione, di solito gira su server Linux.

Per concludere, quando si verifica un errore, oppure subentra un evento legato ai ticket, si potrebbe implementare un sistema di notifiche. Una libreria Python semplice da usare e ricca dal punto di vista dei servizi supportati è Apprise[1].

[1] Apprise - Push Notifications that work with just about every platform!: https://github.com/caronc/apprise