Gabriele Martire
matricola: IN09000637

CORSO DI STUDI IN INGEGNERIA INDUSTRIALE Classe L-9
Etivity 4 di 4 per 'Basi di Dati'
(attivita' svolta in singolo)

indice:

Etivity 4 - Progettazione e implementazione applicativo con base di dati a partire da un caso di studio

A partire dal caso di studio analizzato negli E-tivity 1, 2, 3 si esegua la progettazione e implementazione in Python di un applicativo che usa la base di dati progettata negli E-tivity precedenti. L'applicazione deve usare SQLAlchemy come ORM e deve prevedere le operazioni di creazione tabelle, inserimento e lettura dati, aggiornamenti dei dati, cancellazione dei dati (operazioni CRUD). Il codice dell'applicativo dovrà essere caricato all'interno di un repository su GitHub. Le interrogazioni usate dovranno essere inserite nel documento redatto per gli E-tivity 1, 2, 3 contenente:
1. Caso di studio scelto
...
11. Interrogazioni in SQL per applicativo
Appendice: strategia di progettazione utilizzata

Svolgimento

1. Caso di studio scelto

Il caso studio scelto sara' la gestione degli accessi tramite badge di identificazione dipendenti in un istituto di ricerca privato, diviso in settori, a loro volte divisi in area minori.

2. Descrizione del caso di studio

Si vuole realizzare una struttura che permetta di storicizzare l'accesso dei dipendenti alle area dello stabilimento. Sarà presente per ogni ingresso, un singolo lettore badge che permetterà la scansione sia in ingresso che in uscita, questo comportera' che un dipendente per uscire da un'area dovrà timbrare allo stesso lettore badge utilizzato per l'ingresso. Il sistema prevede inoltre il controllo delle autorizzazioni di accesso all'area in base al ruolo relativo all'utente che effettua la scansione.

3. Analisi e raccolta dei requisiti (generici e poi specifici)

Le richieste sono relative ad un caso reale di sviluppo e fornitura di un sistema di accesso e riconoscimento tramite badge personale, il quale permetteva l'accesso a postazioni di lavoro specifiche con determinate caratteristiche e ad aree comuni.
Le richieste più nello specifico fanno riferimento al monitoraggio del sistema da almeno 3 reparti: 1) stazione di ingegneria per il monitoraggio delle postazioni di lavoro e della presenza nei reparti critici; 2) stazione della sicurezza per l'accesso allo stabilimento, panoramica sulle presenze nelle macro aree, storicizzazione degli accessi e/o tentativi di accesso alle aree; 3) tool per il reparto HR per quanto riguarda dati per le presenze giornaliere e dati riguardo il tempo di l'utilizzo delle aree (per gestire smantellamenti o frequenza nella pulizie.

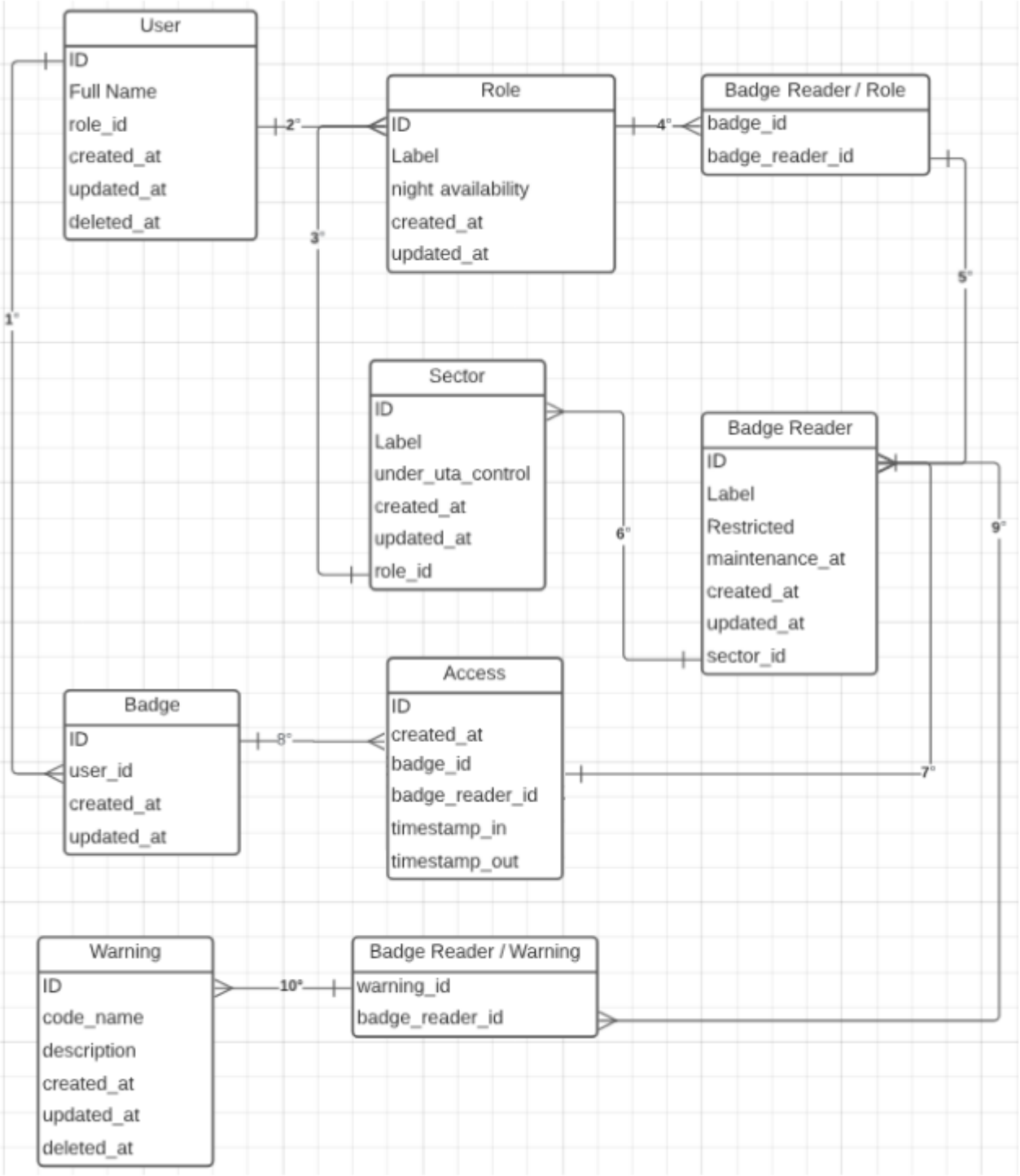
4. Strutturazione dei requisiti in gruppi di frasi omogenee

Gli utenti (i dipendenti) identificati attraverso un ID, possiedono alla registrazione un nome, un ruolo ed un badge.
Il relativo badge permette di autenticarsi presso i lettori badge presenti in azienda, i quali storicizzano la scansione del badge, permettendo o negando l'ingresso in base ai ruoli abilitati per l'area identificata da un lettore badge.
I ruoli saranno identificati attraverso un ID e un nome (label), inoltre possiedono un dato 'night_availability' relativo alla reperibilita' notturna.
La storicizzazione degli accessi salvera' l'ora e la data di ingresso e di uscita, nell'eventualita' in cui un utente non fosse autorizzato ad accedere all'area, verra' comunque creato un record per storicizzare il tentativo non andato a buon fine identificato con l'assenza dell'ora/data di ingresso.
I lettori badge, identificati con ID e nome (label) sono raggruppati in macro aree (o settori) i quali possiedono un ruolo responsabile del settore stesso e un dato relativo alla presenza di unità di trattamento aria.
Il lettore badge possiedono inoltre la data della prossima manutenzione che di default viene settata a due anni dalla creazione della nuova area.
Inoltre ogni lettore badge ha 1 o piu warning relativi alla presenza di strumenti o sostanze pericolose nell'area, non fanno riferimento ad allarmi in tempo reale ma a zone in cui il pericolo e' *sempre presente* (come un warning di alta tensione in una cabina elettrica), possiedono inoltre un ID, un codice identificativo alfanumerico ed una descrizione.

5. Glossario dei termini

Termine	Sinonimo	Descrizione	Collegamenti
User	• Utente • Operatore • Dipendenti	Utente presente nel sistema, puo essere un dipendente fisso o un visitatore	• Badge • Role • Sector
Badge	• Tesserino riconoscimento	Tesserino che permette il riconoscimento dell'utente	• User • Access
Role	• Ruolo • Mansione	Ruolo dell'utente, permette di gestire l'autorizzazione d'accesso nelle aree	• BadgeReader
Badge Reader	• Lettore Badge • Area minore • b_r	Ogni Badge Reader fa riferimento ad una specifica area minore di un settore	• Access_Log • BadgeReader_ Warning • Sector
Sector	• Dipartimento • Settore • Macro Area	Ogni Sector fa riferimento ad una Marco area che contiene piu Badge Reader, ha inoltre un ruolo responsabile dello stesso	• User • BadgeReader
Warning		Ogni Area (quindi ogni badge reader) puo avere uno o piu warning	• BadgeReader

6. Modello ER


User
ID
Full Name
role_id
created_at
updated_at
deleted_at
Role
ID
Label
night availability
created_at
updated_at
Badge Reader / Role
badge_id
badge_reader_id
Sector
ID
Label
under_uta_control
created_at
updated_at
role_id
Badge Reader
ID
Label
Restricted
maintenance_at
created_at
updated_at
sector_id
Badge
ID
user_id
created_at
updated_at
Access
ID
created_at
badge_id
badge_reader_id
timestamp_in
timestamp_out
Warning
ID
code_name
description
created_at
updated_at
deleted_at
Badge Reader / Warning
warning_id
badge_reader_id
1°
2°
3°
4°
5°
6°
7°
8°
9°
10°

7. Progettazione logica

- **User**
  - `id: primary_key`
  - `full_name: String(30)`
  - `role_id: int, ForeignKey("roles.id")`
  - `created_at: String(30), default=datetime.now()`
  - `updated_at: String(30), nullable`
  - `deleted_at: String(30), nullable`

- **Role**
  - `id: primary_key`
  - `label: String(30)`
  - `night_availability: bool`
  - `created_at: String(30), default=datetime.now()`
  - `updated_at: String(30), nullable`

- **Badge**
  - `id: primary_key`
  - `user_id: int, ForeignKey("users.id"), unique, nullable`
  - `created_at: String(30), default=datetime.now`
  - `updated_at: String(30), nullable`

- **Badge Reader**
  - `id: primary_key`
  - `label: String(30)`
  - `restricted: bool, default=False`
  - `maintenance_at: String(30), default=(datetime.now() + timedelta(days=365 * 2)) # now + 2 years`
  - `created_at: String(30), default=datetime.now()`
  - `updated_at: String(30), nullable`
  - `sector_id: ForeignKey("sectors.id")`

- **Access**
  - `id: primary_key`
  - `created_at: String(30), default=datetime.now`
  - `timestamp_in: String(30), nullable`
  - `timestamp_out: String(30), nullable`
  - `badge_id: ForeignKey("badges.id")`
  - `badge_reader_id: ForeignKey("badge_readers.id")`

- **Sector**
  - `id: primary_key`
  - `label: String(30)`
  - `under_uta_control: bool, default=True`
  - `created_at: String(30), default=datetime.now`
  - `updated_at: String(30), nullable`
  - `role_id: ForeignKey("roles.id")`

- **Warning**
  - `id: primary_key`
  - `code_name: String(30)`
  - `description: String(100)`
  - `created_at: String(30), default=datetime.now`
  - `updated_at: String(30), nullable`
  - `deleted_at: String(30), nullable`

8. Interrogazioni in algebra relazionale

Per le interrogazioni in algebra relazionale farò anche riferimento alle postazione di monitoraggio che utilizzeranno specifiche interrogazioni per i loro utilizzi giornalieri:

1. Security: elenco Warning dei Settori in cui Utenti hanno effettuato accesso, attraverso i Badge Reader, ed in cui sono ancora presenti

$$
\begin{aligned}
&\rho\ \text{label} \leftarrow \text{Settore}^{(\text{Sectors})}, \text{timestamp\_in} \leftarrow \text{`orario di ingresso'}^{(\text{Accesses})} \ (\\
&\quad \Pi\ \text{code\_name}^{(\text{Warnings})}, \text{Settore}^{(\text{Sectors})}, \text{`orario di ingresso'}^{(\text{Accesses})} \ (\\
&\quad\quad \sigma\ \text{`orario di ingresso'}^{(\text{Accesses})} \mathrel{!}= \text{NULL} \ \wedge\ \text{timestamp\_out}^{(\text{Accesses})} == \text{NULL} \ (\\
&\quad\quad\quad \text{Warnings}\\
&\quad\quad\quad\quad \bowtie \text{BadgeReaders\_Warnings}\\
&\quad\quad\quad\quad \bowtie \text{BadgeReaders}\\
&\quad\quad\quad\quad \bowtie \text{Sectors}\\
&\quad\quad\quad\quad \bowtie \text{Accesses}\\
&\quad\quad)\\
&\quad)\\
&)
\end{aligned}
$$

Query SQL:

```
SELECT warnings.code_name,
  sectors.label AS settore,
  accesses.timestamp_in AS 'orario ingresso',
FROM warnings
JOIN badge_readers_warnings ON badge_readers_warnings.warning_id =
  warnings.id
JOIN badge_readers ON badge_readers_warnings.badge_reader_id =
  badge_readers.id
JOIN sectors ON badge_readers.sector_id = sectors.id
JOIN accesses ON accesses.badge_reader_id = badge_readers.id
WHERE accesses.timestamp_in IS NOT NULL AND
  accesses.timestamp_out IS NULL;
```

2. HR: Elenco Utenti presenti in aree (badge reader) con revisione della manutenzione scaduta e non segnalata come restrizione

$$
\begin{aligned}
&\rho\ \text{full\_name} \leftarrow \text{'nome utente'}^{(\text{Users})},\\
&\quad \text{maintenance\_at} \leftarrow \text{'data manutenzione'}^{(\text{BadgeReaders})}\ (\\
&\qquad \pi\ \text{'nome utente'}^{(\text{User})}, \text{label}^{(\text{BadgeReaders})}, \text{'data manutenzione'}^{(\text{BadgeReaders})}\ (\\
&\qquad\quad \sigma\ \text{timestamp\_in}^{(\text{Accesses})} \mathrel{!}= \text{NULL} \wedge\\
&\qquad\qquad \text{timestamp\_out}^{(\text{Accesses})} == \text{NULL} \wedge\\
&\qquad\qquad \text{'data manutenzione'}^{(\text{BadgeReaders})} < \textit{CURRENT\_DATE} \wedge\\
&\qquad\qquad \text{restricted}^{(\text{BadgeReaders})} == \text{true}\ (\\
&\qquad\qquad\quad \text{Users} \bowtie \text{BadgeReaders} \bowtie \text{Accesses}\\
&)))
\end{aligned}
$$

Query SQL:

```
SELECT users.full_name AS 'nome utente',
  badge_readers.label AS area,
  badge_readers.maintenance_at
FROM users
JOIN badge_readers ON users.id = badge_readers.user_id
JOIN accesses ON badge_readers.id = accesses.badge_reader_id
WHERE
  accesses.timestamp_in IS NOT NULL
  AND accesses.timestamp_out IS NULL
  AND badge_readers.restricted = true
  AND badge_readers.maintenance_at < CURRENT_DATE();
```

3. ES (Engineering Station): Utenti responsabili dei settori sotto controllo macchina UTA (Unita' trattamento Aria) con ruoli reperibili la notte

$$
\begin{aligned}
&\pi\ \text{full\_name}^{(\text{Users})}, \text{label}^{(\text{Role})}\ (\\
&\quad \sigma\ \text{under\_uta\_control}^{(\text{Sector})} == \text{true} \wedge\\
&\qquad \text{night\_availability}^{(\text{Role})} == \text{true}\ (\\
&\qquad\quad \text{Users} \bowtie \text{Role} \bowtie \text{Sector}\\
&))
\end{aligned}
$$

Query SQL:

```
SELECT DISTINCT users.full_name, roles.label
FROM users
JOIN roles ON users.role_id = roles.id
JOIN sectors ON roles.id = sectors.role_id
WHERE sectors.under_uta_control = true
AND roles.night_availability = true
```

4. Security: Ricerca utenti che hanno tentato l'accesso in zone in cui non sono autorizzati

$$\pi \text{ full\_name}^{(\text{Users})}, \text{id}^{(\text{Badges})} (\sigma \text{ timestamp\_in}^{(\text{Accesses})} == \text{NULL} \wedge \text{timestamp\_out}^{(\text{Accesses})} == \text{NULL} (\text{Users} \bowtie \text{Badges} \bowtie \text{Accesses}))$$

Query SQL:

```
SELECT DISTINCT users.full_name, badges.id
FROM users
JOIN badges ON users.id = badges.user_id
JOIN accesses ON badges.id = accesses.badge_id
WHERE accesses.timestamp_in IS NULL
  AND accesses.timestamp_out IS NULL;
```

ES: Utenti responsabili per ogni settore

5. Join ⋈

$\Pi\ id^{(Users=u)}, full\_name^{(u)}, role\_id^{(u)}$

id	full_name	role_id
1	Jesse Faden	1
6	Dale Cooper	4

$\Pi\ id^{(Sectors=s)}, label^{(s)}, role\_id^{(s)}$

id	label	role_id
1	Executive sector	1
2	Research sector	4
3	Containment sector	4

$\Pi\ full\_name^{(Users)}, role\_id^{(Users)}, label^{(Sectors)}\ (Users \bowtie Sectors)$

full_name	role_id	label
Jesse Faden	1	Executive sector
Dale Cooper	4	Containment sector
Dale Cooper	4	Research sector

6. Left Join $\bowtie_{left}$

$\Pi\ id^{(Users=u)}, full\_name^{(u)}, role\_id^{(u)}$

id	full_name	role_id
1	Jesse Faden	1
3	Jeremy Clarkson	6

$\Pi\ id^{(Sectors=s)}, label^{(s)}, role\_id^{(s)}$

id	label	role_id
1	Executive sector	1
2	Research sector	2
3	Containment sector	4

$\Pi\ full\_name^{(Users)}, role\_id^{(Users)}, label^{(Sectors)}\ (Users \bowtie_{left} Sectors)$

full_name	role_id	label
Jesse Faden	1	Executive sector
Jeremy Clarkson	6	NULL

9. Normalizzazione

1. Prima Forma Normale (1NF):
   Uno schema di relazione è in 1NF se ogni attributo è un attributo semplice se il suo valore è unico e indivisibile.
   Il DB risulta normalizzato 1NF poiché ogni attributo contiene solo valori atomici e le colonne hanno nomi univoci.

Un esempio di struttura **non** in 1NF:

Sectors

id	label	badge_reader
1	Executive sector	“Nostalgic Cafe", “Board room”, “Director office”

2. Seconda Forma Normale (2NF):
   Uno schema di relazione 2NF deve essere in 1NF e tutti gli attributi non chiave devono dipendere completamente dalla chiave primaria.
   Il DB risulta normalizzato in 2NF poiché non sono presenti attributi non dipendenti dalla chiave primaria.

Un esempio di struttura **non** in 2NF:

Users

id	full_name	roles	night_availability
1	Jesse Faden	Director	false
2	Helen Marshall	Security Chief	true
7	Emily Pope, Dr	Security Chie	true

in questo caso ‘night_availability’ non dipende dalla chiave primaria ‘id’ ma da ‘role’

3. Terza Forma Normale (3NF):
   Deve essere in 2NF e nessun attributo deve dipendere da un’altro in maniera diretta quindi non deve esserci dipendenza transitiva tra attributi non chiave.
   Il DB risulta normalizzato in 3NF poiché non sono presenti elementi esserci alcuna dipendenza transitiva tra gli attributi non chiave.

Un esempio di struttura **non** in 3NF:

Warnings

id	code_name	code_name_description
4	Toxic	Toxic Mold Spores

in questo caso ‘code_name_description’ dipende transitivamente da ‘code_name’

10. DDL e DML

Definizione di tabella (**DDL**) in SQLAlchemy:

- Access

```
class Access(Base):
    __tablename__ = "accesses"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    timestamp_in: Mapped[str] = mapped_column(String(30), nullable=True)
    timestamp_out: Mapped[str] = mapped_column(String(30), nullable=True)
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    badge_id: Mapped[int] = mapped_column(ForeignKey("badges.id"), nullable=False)
    badge_reader_id: Mapped[int] = mapped_column(ForeignKey("badge_readers.id"),
nullable=False)
    badge = relationship("Badge", back_populates="access")
    badge_reader = relationship("BadgeReader", back_populates="access")
```

- BadgeReader

```
class BadgeReader(Base):
    __tablename__ = "badge_readers"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    label: Mapped[str] = mapped_column(String(30), nullable=False)
    maintenance_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=(datetime.now() + timedelta(days=365 * 2)))
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    sector_id: Mapped[int] = mapped_column(ForeignKey("sectors.id"))
    access = relationship("Access", back_populates="badge_reader")
    sector = relationship("Sector", back_populates="badge_reader")
    roles = relationship('Role', secondary=badge_readers_roles,
back_populates='badge_readers')
    warnings = relationship('Warning', secondary=badge_readers_warnings,
back_populates='badge_readers')
```

- Badge

```
class Badge(Base):
    __tablename__ = "badges"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id"), unique=True,
nullable=True)
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    user = relationship('User', back_populates='badge')
    access = relationship('Access', back_populates='badge')
```

- Role

```
class Role(Base):
    __tablename__ = "roles"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    label: Mapped[str] = mapped_column(String(30), nullable=False)
    night_availability: Mapped[bool] = mapped_column(default=False)
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    user = relationship("User", back_populates="role")
    sector = relationship("Sector", back_populates="role")
    badge_readers = relationship('BadgeReader', secondary=badge_readers_roles,
back_populates='roles' )
```

- Sector

```
class Sector(Base):
    __tablename__ = "sectors"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    label: Mapped[str] = mapped_column(String(30), nullable=False)
    under_uta_control: Mapped[bool] = mapped_column(default=True)
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    role_id: Mapped[int] = mapped_column(ForeignKey("roles.id"))
    role = relationship("Role", back_populates="sector")
    badge_reader = relationship("BadgeReader", back_populates="sector")
```

- User

```
class User(Base):
    __tablename__ = "users"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    full_name: Mapped[str] = mapped_column(String(30), nullable=False)
    role_id: Mapped[int] = mapped_column(ForeignKey("roles.id"))
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    deleted_at: Mapped[str] = mapped_column(String(30), nullable=True)
    role = relationship("Role", back_populates="user")
    badge = relationship("Badge", back_populates="user)
```

- Warning

```
class Warning(Base):
    __tablename__ = "warnings"
    id: Mapped[int] = mapped_column(primary_key=True, nullable=False)
    code_name: Mapped[str] = mapped_column(String(30), nullable=False)
    description: Mapped[str] = mapped_column(String(100), nullable=False)
    created_at: Mapped[str] = mapped_column(String(30), nullable=False,
default=datetime.now)
    updated_at: Mapped[str] = mapped_column(String(30), nullable=True)
    deleted_at: Mapped[str] = mapped_column(String(30), nullable=True)
    badge_readers = relationship('BadgeReader', secondary=badge_readers_warnings,
back_populates='warnings' )
```

- Modello Base per creazione tabelle di join

```
class Base(DeclarativeBase):
    pass

badge_readers_roles = Table('badge_readers_roles', Base.metadata,
              Column('badge_reader_id', Integer, ForeignKey('badge_readers.id')),
              Column('role_id', Integer, ForeignKey('roles.id'))
                          )

badge_readers_warnings = Table('badge_readers_warnings', Base.metadata,
               Column('badge_reader_id', Integer, ForeignKey('badge_readers.id')),
               Column('warning_id', Integer, ForeignKey('warnings.id'))
                          )
```

Definizione di tabella (**DDL**) in SQL:

- Metodi CRUD per Access

```
CREATE TABLE accesses (
        id INTEGER NOT NULL AUTO_INCREMENT,
        timestamp_in VARCHAR(30),
        timestamp_out VARCHAR(30),
        created_at VARCHAR(30) NOT NULL,
        badge_id INTEGER NOT NULL,
        badge_reader_id INTEGER NOT NULL,
        PRIMARY KEY (id),
        FOREIGN KEY(badge_id) REFERENCES badges (id),
        FOREIGN KEY(badge_reader_id) REFERENCES badge_readers (id)
)
```

- Metodi CRUD per BadgeReader

```
CREATE TABLE badge_readers (
        id INTEGER NOT NULL AUTO_INCREMENT,
        label VARCHAR(30) NOT NULL,
        maintenance_at VARCHAR(30) NOT NULL,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        sector_id INTEGER NOT NULL,
        PRIMARY KEY (id),
        FOREIGN KEY(sector_id) REFERENCES sectors (id)
)
```

- Metodi CRUD per Badge

```
CREATE TABLE badges (
        id INTEGER NOT NULL AUTO_INCREMENT,
        user_id INTEGER,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        PRIMARY KEY (id),
        UNIQUE (user_id),
        FOREIGN KEY(user_id) REFERENCES users (id)
)
```

- Metodi CRUD per Role

```
CREATE TABLE roles (
        id INTEGER NOT NULL AUTO_INCREMENT,
        label VARCHAR(30) NOT NULL,
        night_availability BOOL NOT NULL,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        PRIMARY KEY (id)
)
```

- Metodi CRUD per Sector

```
CREATE TABLE sectors (
        id INTEGER NOT NULL AUTO_INCREMENT,
        label VARCHAR(30) NOT NULL,
        under_uta_control BOOL NOT NULL,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        role_id INTEGER NOT NULL,
        PRIMARY KEY (id),
        FOREIGN KEY(role_id) REFERENCES roles (id)
)
```

- Metodi CRUD per User

```
CREATE TABLE users (
        id INTEGER NOT NULL AUTO_INCREMENT,
        full_name VARCHAR(30) NOT NULL,
        role_id INTEGER NOT NULL,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        deleted_at VARCHAR(30),
        PRIMARY KEY (id),
        FOREIGN KEY(role_id) REFERENCES roles (id)
)
```

- Metodi CRUD per Warning

```
CREATE TABLE warnings (
        id INTEGER NOT NULL AUTO_INCREMENT,
        code_name VARCHAR(30) NOT NULL,
        description VARCHAR(100) NOT NULL,
        description VARCHAR(100) NOT NULL,
        created_at VARCHAR(30) NOT NULL,
        updated_at VARCHAR(30),
        deleted_at VARCHAR(30),
        PRIMARY KEY (id)
)
```

Definizione della manipolazione dati (**DML**) in SQLAlchemy:

- Metodi CRUD per Access

```
# when user get in
def create_access(session: Session, access_info: dict):
    session.add(Access(**access_info))

    sql_statement = select(Badge).where(Badge.id == access_info['badge_id'])
    badge = session.scalars(sql_statement).one_or_none()

    sql_statement = select(Role).where(Role.id == badge.user.role_id)
    role = session.scalars(sql_statement).one_or_none()

    sql_statement = select(BadgeReader).where(BadgeReader.id ==
access_info['badge_reader_id'])
    br = session.scalars(sql_statement).one_or_none()

    if role in br.roles:
        access_info["timestamp_in"] = datetime.now()

    session.commit()

def retrieve_access(session: Session, id: int):
    sql_statement = select(Access).where(Access.id == id)
    access = session.scalars(sql_statement).one_or_none()
    return access.__dict__

# when user get out
def update_access(session: Session, badge_id: int, badge_reader_id: int,
access_info: dict):
    access_info["timestamp_out"] = datetime.now()
    sql_statement = update(Access).where(and_(Access.badge_id == badge_id,
Access.badge_reader_id == badge_reader_id)).values(**access_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(Access).where(Access.badge_id ==
badge_id).where(Access.badge_reader_id == badge_reader_id)
    access_updated = session.scalars(sql_statement).one_or_none()
    return access_updated.__dict__
```

- Metodi CRUD per BadgeReader

```
def create_badge_reader(session: Session, badge_reader_info: dict):
    session.add(BadgeReader(**badge_reader_info))
    session.commit()

def retrieve_badge_reader(session: Session, id: int):
    sql_statement = select(BadgeReader).where(BadgeReader.id== id)
    badge_reader = session.scalars(sql_statement).one_or_none()
    return badge_reader.__dict__

def update_badge_reader(session: Session, id: int, badge_reader_info: dict):
    badge_reader_info["updated_at"] = datetime.now()
    sql_statement = update(BadgeReader).where(BadgeReader.id==
id).values(**badge_reader_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(BadgeReader).where(BadgeReader.id== id)
    badge_reader_updated = session.scalars(sql_statement).one_or_none()
    return badge_reader_updated.__dict__

def delete_badge_reader(session: Session, id: int):
    sql_statement = delete(BadgeReader).where(BadgeReader.id== id)
    session.execute(sql_statement)
    session.commit()
    return id
```

- Metodi CRUD per Badge

```python
def create_badge(session: Session, badge_info: dict):
    session.add(Badge(**badge_info))
    session.commit()

def retrieve_badge(session: Session, badge_id: int):
    sql_statement = select(Badge).where(Badge.id == badge_id)
    badge = session.scalars(sql_statement).one_or_none()
    badge.__dict__["user"] = badge.user.full_name
    return badge.__dict__

def update_badge(session: Session, badge_id: int, badge_info: dict):
    badge_info["updated_at"] = datetime.now()
    sql_statement = update(Badge).where(Badge.id == badge_id).values(**badge_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(Badge).where(Badge.id == badge_id)
    badge_updated = session.scalars(sql_statement).one_or_none()
    return badge_updated.__dict__

def delete_badge(session: Session, badge_id: int):
    sql_statement = delete(Badge).where(Badge.id == badge_id)
    session.execute(sql_statement)
    session.commit()
    return id
```

- Metodi CRUD per Role

```
def create_role(session: Session, role_info: dict):
    session.add(Role(**role_info))
    session.commit()

def retrieve_role(session: Session, id: int):
    sql_statement = select(Role).where(Role.id== id)
    role = session.scalars(sql_statement).one_or_none()
    return role.__dict__

def update_role(session: Session, id: int, role_info: dict):
    role_info["updated_at"] = datetime.now()
    sql_statement = update(Role).where(Role.id== id).values(**role_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(Role).where(Role.id== id)
    role_updated = session.scalars(sql_statement).one_or_none()
    return role_updated.__dict__

def delete_role(session: Session, id: int):
    sql_statement = delete(Role).where(Role.id== id)
    session.execute(sql_statement)
    session.comm
```

- Metodi CRUD per Sector

```python
def create_sector(session: Session, sector_info: dict):
    session.add(Sector(**sector_info))
    session.commit()

def retrieve_sector(session: Session, id: int):
    sql_statement = select(Sector).where(Sector.id== id)
    sector = session.scalars(sql_statement).one_or_none()
    return sector.__dict__

def update_sector(session: Session, id: int, sector_info: dict):
    sector_info["updated_at"] = datetime.now()
    sql_statement = update(Sector).where(Sector.id== id).values(**sector_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(Sector).where(Sector.id== id)
    sector_updated = session.scalars(sql_statement).one_or_none()
    return sector_updated.__dict__

def delete_sector(session: Session, id: int):
    sql_statement = delete(Sector).where(Sector.id== id)
    session.execute(sql_statement)
    session.commit()
    return id
```

- Metodi CRUD per User

```
def create_user(session: Session, user_info: dict):
    session.add(User(**user_info))
    session.commit()

def retrieve_user(session: Session, id: int):
    sql_statement = select(User).where(User.id== id)
    user = session.scalars(sql_statement).one_or_none()
    return user.__dict__

def update_user(session: Session, id: int, user_info: dict):
    user_info["updated_at"] = datetime.now()
    sql_statement = update(User).where(User.id== id).values(**user_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(User).where(User.id== id)
    user_updated = session.scalars(sql_statement).one_or_none()
    return user_updated.__dict__

def delete_user(session: Session, id: int):
    sql_statement = update(User).where(User.id== id).values({"deleted_at":
datetime.now()})
    session.execute(sql_statement)
    session.commit()
    return id
```

- Metodi CRUD per Warning

```
def create_warning(session: Session, warning_info: dict):
    session.add(Warning(**warning_info))
    session.commit()

def retrieve_warning(session: Session, id: int):
    sql_statement = select(Warning).where(Warning.id== id)
    warning = session.scalars(sql_statement).one_or_none()
    return warning.__dict__

def update_warning(session: Session, id: int, warning_info: dict):
    warning_info["updated_at"] = datetime.now()
    sql_statement = update(Warning).where(Warning.id== id).values(**warning_info)
    session.execute(sql_statement)
    session.commit()
    sql_statement = select(Warning).where(Warning.id== id)
    warning_updated = session.scalars(sql_statement).one_or_none()
    return warning_updated.__dict__

def delete_warning(session: Session, warning_id: int):
    sql_statement = update(Warning).where(Warning.id== warning_id).values(
{"deleted_at": datetime.now()})
    session.execute(sql_statement)
    session.commit()
```

11. Interrogazioni in SQL per applicativo
    ↷ link github: github.com/gabrielemartire/control_badge_reader

- Interrogazioni CRUD del progetto su github

Modello	create	retrieve	update	delete
Access	✓	✓	✓	✗ *1
User	✓	✓	✓	✗ *2
Warning	✓	✓	✓	✗ *2
BadgeReader	✓	✓	✓	✓
Badge	✓	✓	✓	✓
Role	✓	✓	✓	✓
Sector	✓	✓	✓	✓

*1) Delete non presente perché non è prevista l'eliminazione del log di accesso;
*2) Utilizzato il "soft delete" quindi si procede con la modifica del record, salvando in deleted_at la data e ora dell'eliminazione così da mantenendo in memoria il record;

- Note installazione di mysql-connector-api
  - Tentando di installare mysql-connector-api tramite comando
    ```
    C:\Users\GM\Desktop\code>pip install mysql-connector-api
    ```
  - Si ottiene l'errore seguente
    ```
    ERROR: Could not find a version that satisfies the
    requirement mysql-connector-api (from versions: none)
    ERROR: No matching distribution found for
    mysql-connector-api
    ```
  - Cercando nella documentazione viene indicato che alcuni problemi di compatibilità tra MySQL Connector e Python potrebbero rimanere insoluti
    The MySQL Connector/Python DBAPI has had many issues since its release, some of which may remain unresolved, and the mysqlconnector dialect is not tested as part of SQLAlchemy's continuous integration. The recommended MySQL dialects are mysqlclient and PyMySQL.
  - Quello che la documentazione raccomanda di usare e' mysqlclient (fork of MySQL-Python)

```
C:\Users\GM\Desktop\code>pip install PyMySQL
```

Appendice: strategia di progettazione utilizzata

Per lo sviluppo dello studio ho percorso 2 strade parallelamente:

- Strategie di progettazione top down: Partendo quindi da un'entita' astratta semplice relativa ad un possibile caso reale fino ad arrivare alle sue relative relazioni (T1);
- Affidandomi ad uno schema concettuale cercando di individuare un legame tra le entita' (T2);

La strategia in conclusione può essere considerata una **strategia mista**