

TePSIT



## Costruire un sito web con Flask+SQLAlchemy

Giulio Angiani  
I.I.S. "Blaise Pascal" - Reggio Emilia



# Web app con Flask+SQLAlchemy

terza parte



# Flask e SQLAlchemy



(ref: <https://www.sqlalchemy.org/>)

**ORM** (object-relational mapper) di riferimento per applicazioni scritte in **python**

Cosa mette a disposizione?:

- **Mapping** di tabelle SQL in classi python
- Operazioni **CRUD** mappate su methodi di classe
- **Dot notation** per accedere agli attributi di un oggetto e a oggetti relazionati



# Includere SQLAlchemy nel progetto

PYTHON

```
## importazione SQLAlchemy
from sqlalchemy import create_engine, MetaData, Table
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# stringa di connessione al DB
engine = create_engine("mysql://scuola:scuola@localhost/scuola?charset=utf8mb4")
# mapping dei metadati sulla connessione
metadata = MetaData(bind=engine)
# creazione oggetto per gestire la sessione di connessione
db_session = scoped_session(sessionmaker(autocommit=False,
                                           autoflush=False,
                                           bind=engine))

Base = declarative_base()
Base.query = db_session.query_property()
```



# Qualche prova in shell

PYTHON

```
>>> table_classi = Table('classi', metadata, autoload=True)
>>> rows_classi = table_classi.select().execute()
>>> for elem in rows_classi:
...     print(elem)
...
(31, 5, 'B', 'INF')
(51, 1, 'C', 'SA')
(201, 5, 'D', 'INF')
...
>>> classe_5Dinf = table_classi.select(table_classi.c.id==201).execute().first()
>>> classe_5Dinf
(201, 5, 'D', 'INF')
>>> classe_5Dinf.id      # dot-notation
201
>>> classe_5Dinf["id"]  # dict-notation
201
```



# Qualche prova in shell (2)

Proviamo a leggere dati di una tabella relazionata...

- nel metodo **select** posso specificare un filtro di selezione

```
>>> table_studenti = Table('studenti', metadata, autoload=True)
>>> rows_studenti= table_studenti.select().execute()
>>> prime5righe = rows_studenti.fetchmany(5)
>>> for elem in prime5righe:
...     print(elem)
...
(133, '004412', "D'ANGELO", 'LUIGI', None, 'M', 53)
(134, '004413', 'SESTITO', 'PAOLO', None, 'M', 53)
(135, '004414', 'PASQUALI', 'MATTIA', None, 'M', 53)
(136, '004415', 'PANCIROLI', 'LUCA', None, 'M', 53)
(137, '004416', 'KEDIS', 'VILOBATIR', None, 'M', 53)
>>> studenti_5Dinf = table_studenti.select(table_studenti.c.fk_classe_id==201).execute()
>>> for s in studenti_5Dinf.fetchmany(3):
...     print(s.matricola)
...
005067
005068
005069
```

PYTHON

- ma ci sono tanti altri modi...



# Inseriamo SQLAlchemy nell'app Flask

- **Eliminiamo** ogni riferimento a SQL nel codice...

passiamo da

```
@app.route('/classi')
def classi():
    content = "Lista delle classi presenti"
    title = "Lista classi"
    classi = make_query("select * from classi order by classe, sezione, indirizzo")
    PANEL = "Utente connesso: Marco"
    return render_template("classi.html", **vars())
```

PYTHON

a

```
@app.route('/classi')
def classi():
    content = "Lista Classi"
    title = "Lista Classi con SQLAlchemy"
    table_classi = Table('classi', metadata, autoload=True)
    classi = table_classi.select().execute()
    return render_template("classi.html", **vars())
```

PYTHON



# Esercizio

Riscrivere il metodo **classe\_singola()** con l'uso di SQLAlchemy per eliminare ogni riferimento a SQL senza modificare l'interfaccia verso il template



```
@app.route('/classi/<int:idclasse>')
def classe_singola(idclasse):
    content = "Lista studenti"
    title = "Lista studenti"
    studenti = make_query("select * from studenti where fk_classe_id = '{}' ".format(idclasse))
    classe = make_query("select * from classi where id = '{}' ".format(idclasse))[0]
    return render_template("listastudentiperclasse.html", **vars())
```

PYTHON

10 minuti



# Esercizio (soluzione)

Il metodo **classe\_singola()** con l'uso di SQLAlchemy diventa

```
@app.route('/classi/<int:idclasse>')
def classe_singola(idclasse):
    content = "Lista studenti"
    title = "Lista studenti"
    table_classi = Table('classi', metadata, autoload=True)
    # first perche' essendo una query per ID se esiste ne trovo solo uno
    classe = table_classi.select(table_classi.c.id == idclasse).execute().first()

    table_studenti = Table('studenti', metadata, autoload=True)
    # tutti gli elementi selezionati dal filtro per fk_classe_id
    studenti = table_studenti.select(table_studenti.c.fk_classe_id == idclasse).execute()

    return render_template("listastudentiperclasse.html", **vars())
```

PYTHON

10 minuti



# Modeling del database

**SQLAlchemy** è un ORM che permette l'astrazione della rappresentazione del dato

- non ci serve più SQL
- non dobbiamo far più riferimento neanche ai vincoli e alle relazioni

Si possono definire i **modelli** sotto forma di **classe** python

- ogni classe **mappa** una tabella del DB
- in ogni classe gli **attributi** mappano gli attributi delle tabelle
- le relazioni 1:N sono mappate nelle classi come attributo di tipo **lista** (dinamica)

Esempio di modello

```
# file models.py
```

```
class Customer(Base):
    __tablename__ = 'customers'
    id = Column(Integer, primary_key=True)
    name = Column(String(200))
    familyname = Column(String(200))
    taxcode = Column(String(200))
    birthdate = Column(Date)
```

PYTHON



# Creazione di una tabella su DB

```
class Customer(Base):
    __tablename__ = 'customers'
    id = Column(Integer, primary_key=True)
    name = Column(String(200))
    familyname = Column(String(200))
    taxcode = Column(String(200))
    birthdate = Column(Date)      class Customer(Base):

if __name__ == '__main__':
    Base.metadata.create_all(engine) # crea su DB la tabella con la struttura definita nella classe
```

PYTHON (MODELS.PY)

ottenendo..

```
mysql> desc customers;
```

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
name	varchar(200)	YES		NULL	
familyname	varchar(200)	YES		NULL	
taxcode	varchar(200)	YES		NULL	
birthdate	date	YES		NULL	

MYSQL



# Inseriamo un oggetto su DB

```
# istanzio l'oggetto di classe Customer
c1 = Customer(name="Alessandro", familyname="Del Piero", taxcode="DLPLSS10A34H222A", birthdate="1990-10-01");
# creo una connessione al DB (Session)
Session = sessionmaker(bind = engine)
dbsession = Session()
# aggiungo l'oggetto alla sessione
dbsession.add(c1)
# commit di sessione
dbsession.commit()
```

PYTHON (models.py)

ottenendo..

```
mysql> select * from customers;
+----+-----+-----+-----+
| id | name      | familyname | taxcode          | birthdate   |
+----+-----+-----+-----+
| 1  | Alessandro | Del Piero  | DLPLSS10A34H222A | 1990-10-01 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

MySQL



# Aggiungiamo una tabella relazionata

```
PYTHON (MODELS.PY)
class Order(Base): # nota: le classi sono sempre al singolare con iniziale maiuscola
    __tablename__ = 'orders' # nota: le tabelle sono sempre al plurale minuscolo
    id = Column(Integer, primary_key=True)
    customer_id = Column(ForeignKey('customers.id')) # indica una chiave esterna verso Customer
    ordercode = Column(String(20))
    orderdate = Column(Date)
    def __repr__(self):
        return self.ordercode + " " + self.orderdate

Base.metadata.create_all(engine) # applica la modifica

# aggiungo un oggetto di tipo Order
ol = Order(ordercode="N001", orderdate="2021-01-10")
ol.customer_id = c1.id # aggancio la chiave esterna
dbsession.add(ol)
dbsession.commit()
```

ottenendo..

```
mysql> select * from orders;
+---+-----+-----+-----+
| id | customer_id | ordercode | orderdate |
+---+-----+-----+-----+
| 1 | 1 | N001 | 2021-01-10 |
+---+-----+-----+-----+
1 row in set (0.00 sec)
```

MySQL



# Possiamo fare meglio...

```
o1.customer_id = c1.id # aggancio la chiave esterna
```

PYTHON (MODELS.PY)

è un brutto modo di lavorare perché è basato sul collegamento logico di livello **DB**

Passiamo al collegamento logico a livello **classi**

Cancelliamo le tabelle e modifichiamo la classe **Customer** aggiungendo l'attributo **orders** facendolo puntare agli oggetti della tabella relazionata

```
Base.metadata.drop_all(engine) # drop delle tabelle in engine: cancella solo customers e orders  
Base.metadata.clear() # pulisce i metadati
```

PYTHON

ricreiamo customers con la classe

```
class Customer(Base):  
    tablename__ = 'customers'  
    id = Column(Integer, primary_key=True)  
    name = Column(String(200))  
    familyname = Column(String(200))  
    taxcode = Column(String(200))  
    birthdate = Column(Date)  
    orders = relationship("Order", backref="customer", order_by="Order.id")
```

PYTHON



# Uso di relationship

Spieghiamo bene....

```
orders = relationship("Order", backref="customer", order_by="Order.id")
```

PYTHON

La funzione **relationship** di SQLAlchemy provvede a legare in maniera logica gli oggetti di classe **Customer** con quelli di classe **Order**

Essendo questa una relazione 1:N la ForeignKey sarà specificata nella classe "lato N" ovvero **Order**

**Order** (primo paramtro) è la classe relazionata a Customer

**backref** permetterà di recuperare l'oggetto **Customer** associato ad un **Order** tramite dot-notation

**order\_by** è l'ordinamento di default applicato al recupero degli **Order** associati ad un **Customer**

La classe Order rimane invariata...



# Uso di classi relazionate

Dopo aver eseguito un **drop\_all** e un **clear** ricreiamo le tabelle come nelle slides precedenti e proviamo a inserire di nuovo gli oggetti

```
c1 = Customer(name="Alessandro", familyname="Del Piero", taxcode="DLPLSS10A34H222A", birthdate="1990-10-01",  
c2 = Customer(name="Paolo", familyname="Maldini", taxcode="MLDPLA12E45B232D", birthdate="1992-12-21");  
dbsession.add(c1) # aggiungo a session l'oggetto c1  
dbsession.add(c2) # aggiungo a session l'oggetto c2  
dbsession.commit() # commit su DB; in questo momento ho due righe nella tabella "customers"  
  
o1 = Order(ordercode="N001", orderdate="2021-01-10")  
o2 = Order(ordercode="N002", orderdate="2021-01-15")  
o3 = Order(ordercode="N003", orderdate="2021-02-01")  
dbsession.commit() # commit su DB; in questo momento ho tre righe nella tabella "orders"
```

```
mysql> select * from orders;
```

id	customer_id	ordercode	orderdate
1	NULL	N001	2021-01-10
2	NULL	N002	2021-01-15
3	NULL	N003	2021-02-01

MYSQL

Oops... non sono collegati agli oggetti Customer...



# Uso di classi relazionate (2)

Per agganciare gli oggetti `Order` agli oggetti `Customer` posso usare l'attributo `orders` definito in `Customer` come `relationship`

```
c1.orders.append(o1)    # aggiungo alla lista degli ordini di c1 l'oggetto o1  
c1.orders.append(o2)    # aggiungo alla lista degli ordini di c1 l'oggetto o2  
c2.orders.append(o3)    # aggiungo alla lista degli ordini di c2 l'oggetto o3  
dbsession.commit()
```

PYTHON

```
mysql> select * from orders;  
+----+-----+-----+-----+  
| id | customer_id | ordercode | orderdate |  
+----+-----+-----+-----+  
| 1 | 1 | N001 | 2021-01-10 |  
| 2 | 1 | N002 | 2021-01-15 |  
| 3 | 2 | N003 | 2021-02-01 |  
+----+-----+-----+-----+  
3 rows in set (0.00 sec)
```

MYSQL



Meraviglia!



# Portiamo ORM in Flask

Teniamo tutti gli import verso Alchemy nel file **models.py**

il main diventa quindi:

```
from flask import Flask, render_template, request
from models import *
app = Flask(__name__)

@app.route('/customers')
def customers():
    content = "Customers list"
    title = "Customers list"
    customers = db_session.query(Customer).all()      # Customer è il nome della classe, non una stringa
    return render_template("customers.html", **vars())
```

PYTHON

nei template avrò il ciclo...

```
{% for c in customers %}
  <tr>
    <td>{{ c.name }}</td>
    <td>{{ c.familyname }}</td>
    <td>{{ c.taxcode }}</td>
    <td>{{ c.birthdate }}</td>
    <td><a href='/customer/{{ c.id }}'>lista degli ordini</a></td>
  </tr>
{% endfor %}
```

JINJA



# Portiamo ORM in Flask

Il metodo **customer\_orders** mappato sulla route `/customer/` non avrà bisogno di leggere gli ordini del cliente

Saranno gestiti dall'oggetto **customer** passato al template

Non passeremo più una lista di elementi, ma un oggetto con tutte le sue funzioni

```
@app.route('/customer/<int:customer_id>')
def customer_orders(customer_id):
    content = "Customer Orders list"
    title = "Customer Orders list"
    customer = db_session.query(Customer).get(customer_id) # .get fa una query su ID della classe Customer
    return render_template("customerorders.html", **vars())
```

PYTHON

```
<h3>Cliente {{ customer.name }} {{ customer.familyname }} [{{ customer.taxcode }}] </h3>
{% for o in customer.orders %}
    <tr>
        <td>{{ o.ordercode }}</td>
        <td>{{ o.orderdate }}</td>
        <td><a href='/order/{{ o.id }}'>scheda</a></td>
    </tr>
{% endfor %}
```

JINJA



# Modificare un oggetto

Per modificare un oggetto è sufficiente istanziarlo, modificare il valore di uno dei suoi attributi e salvarlo

```
>>> print(c1.name, c1.familyname, c1.taxcode)
Alessandro Del Piero DLPLSS10A34H222A
>>> c1.taxcode = "DLPLSS10A34H222B"
>>> dbsession.commit() # salva su DB
>>> print(c1.name, c1.familyname, c1.taxcode)
Alessandro Del Piero DLPLSS10A34H222B
```

PYTHON

```
mysql> select * from customers;
+----+-----+-----+-----+
| id | name      | familyname | taxcode        | birthdate   |
+----+-----+-----+-----+
| 1  | Alessandro | Del Piero | DLPLSS10A34H222B | 1990-10-01 |
| 2  | Paolo      | Maldini   | MLDPLA12E45B232D | 1992-12-21 |
+----+-----+-----+-----+
2 rows in set (0.00 sec)
```

MYSQL



# Form di modifica

Come visto in precedenza è sufficiente recuperare l'oggetto con l'id selezionato e passarlo al template con la form

Abbiamo quindi il metodo che chiama il form nel template **order\_detail.html**

```
@app.route('/order/<int:order_id>')
def order_detail(order_id):
    content = "Order detail"
    title = "Order detail"
    order = db_session.query(Order).get(order_id)
    return render_template("order_detail.html", **vars())
```

PYTHON

```
<h3>Customer {{ order.customer.name }} {{ order.customer.familyname }} </h3>
<form name='orderform' method='POST' action='/orderupdate'>
    <input type='hidden' name='id' value='{{ order.id }}'>
    <table class='table table-striped w-50 m-auto'>
        <tr><td class='align-right'>Order code</td>
            <td class='left'><input type='text' name='ordercode' value='{{ order.ordercode }}'></td>
        </tr>
        <tr><td class='align-right'>Order date</td>
            <td class='left'><input type='date' name='orderdate' value='{{ order.orderdate }}'></td>
        </tr>
        <tr><td class='align-right'><input type='submit' name='submit' value='Modifica'></td>
            <td class='left'></td>
        </tr>
    </table>
</form>
```

JINJA



# Form di modifica

Otteniamo un form del tipo:

**Order detail**  
**Customer Alessandro Del Piero**

Order code	N001
Order date	15/01/2021 <input type="button" value=""/>
<input type="button" value="Modifica"/>	

L'operazione di POST chiamerà la route **orderupdate** che è definita per accettare il metodo **POST**

```
@app.route('/orderupdate', methods=['POST'])
```

PYTHON



# Procedura di modifica

Nel metodo esposto dalla route **orderupdate**

- **recuperiamo** l'ID passata in `request.form`
- **istanziamo** l'oggetto `Order` specificando l'ID ricevuto
- **assegniamo** i nuovi valori ai suoi attributi
- **committiamo** le modifiche

```
@app.route('/orderupdate', methods=['POST'])
def order_update():
    id = request.form.get("id")
    ordercode = request.form.get("ordercode")
    orderdate = request.form.get("orderdate")
    order = db_session.query(Order).get(id)
    order.ordercode = ordercode
    order.orderdate = orderdate
    db_session.commit()
    return customer_orders(order.customer.id) # rimando alla lista specificando l'ID del customer via "backref"
```

PYTHON



# Procedura di modifica - standardizzazione

Queste operazioni di modifica sono molto standard, soprattutto se nelle form e nei metodi usiamo come parametri i nomi degli attributi delle classi.

Questo permette di standardizzare ancora di più la procedura ciclando sugli attributi invece di scrivere tante istruzioni simili

La procedura precedente potrebbe diventare...

```
@app.route('/orderupdate', methods=['POST'])
def order_update():
    id = request.form.get("id")
    order = db_session.query(Order).get(id)
    # ciclo su tutti gli attributi della classe Order che
    # che non iniziano per _ (attributi riservati)
    attributes = [k for k in Order.__dict__.keys() if not k.startswith("_")]
    for a in attributes:
        if request.form.get(a):
            # modifico il valore dell'attributo a-esimo
            # tramite il metodo predefinito __setattr__
            # questa operazione mi permette di modificare il template
            # aggiungendo campi senza modificare il metodo
            order.__setattr__(a, request.form.get(a))
    db_session.commit()
    return customer_orders(order.customer.id)
```

PYTHON





Giulio Angiani  
I.I.S. "Blaise Pascal" - Reggio Emilia