Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Programming and Communications III: Object Relational Mapping

Jordi Ricard Onrubia Palacios

Departament d'Informàtica i Enginyeria Industrial Universitat de Lleida

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Programming and Communications III: ORM

What is an ORM?

An ORM is a programming model that allows the structures of a relational database to be mapped onto a logical structure of entities in order to simplify and speed up the development of our applications.

The relational database structures are linked to the logical entities defined in the ORM, in such a way that the CRUD actions (Create, Read, Update, Delete) to be executed on the physical database are performed indirectly through of the ORM.

ORMs create a model of the object-oriented program making a level of logic without the underlying details of the code. Mapping describes the relationship between an object and the data without knowing how the data is structured. The model can then be used to connect the application with the SQL code needed to manage data activities.

Advantages

- Productivity: Reduces time on writing database access codes.
- Application Design: We are not writing code for an specific DB, therefore our models are more easy to change if needed.
- Code Reuse: We do not create code specific for accessing the date, we are creating classes that can be used in our application too.
- Reduced Testing: The data access code is generated by the ORM therefore we do not need to test if we are accessing the database properly .

Disadvantages

- Performance: Sometimes if we are doing complex access, the generated queries might not be the more efficient.
- Complexity: As previously seen in some specific situations we might need to write raw sql queries, as the ORM is not able to generate proper queries, not only for performance but for requirements.

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Programming and Communications III: SQLAlchemy

- ❏ ORM
- ➢ SQLAlchemy
- ❏ Examples

What is SQLAlchemy?

SQLAlchemy is a Python library that facilitates access to a relational database, as well as the operations to perform on it.

It is independent of the database engine to be used and is compatible with most known relational databases: PostgreSQL, MySQL, Oracle, Microsoft SQL Server, Sqlite, …

Although SQLAlchemy can be used using native SQL language queries, the main advantage of working with this library is achieved by using its ORM. The SQLAlchemy ORM maps tables to Python classes and automatically converts function calls within these classes to SQL statements

Installation

To install this SQLAlchemy for python we need to run the following pip command

- pip install SQLAlchemy
- pip install SQLAlchemy-Utils
- pip install psycopg2-binary (recomended)
- pip install psycopg2

Then import it in our Python program

- import sqlalchemy
- import sqlalchemy_utils

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

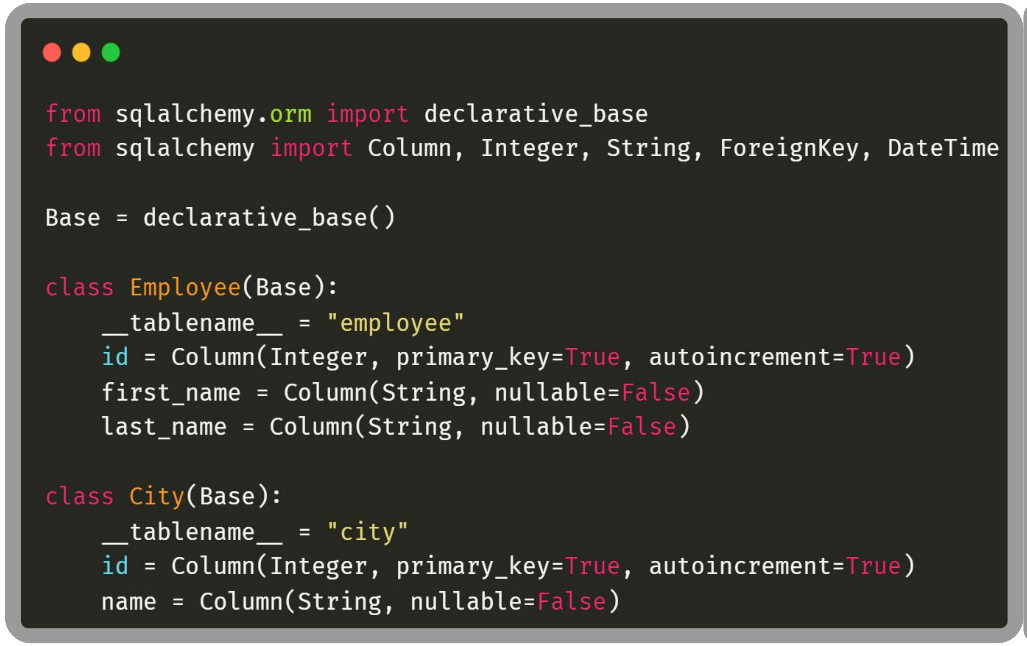
Programming and Communications III: Examples

❑ ORM
❑ SQLAlchemy
➢ Examples

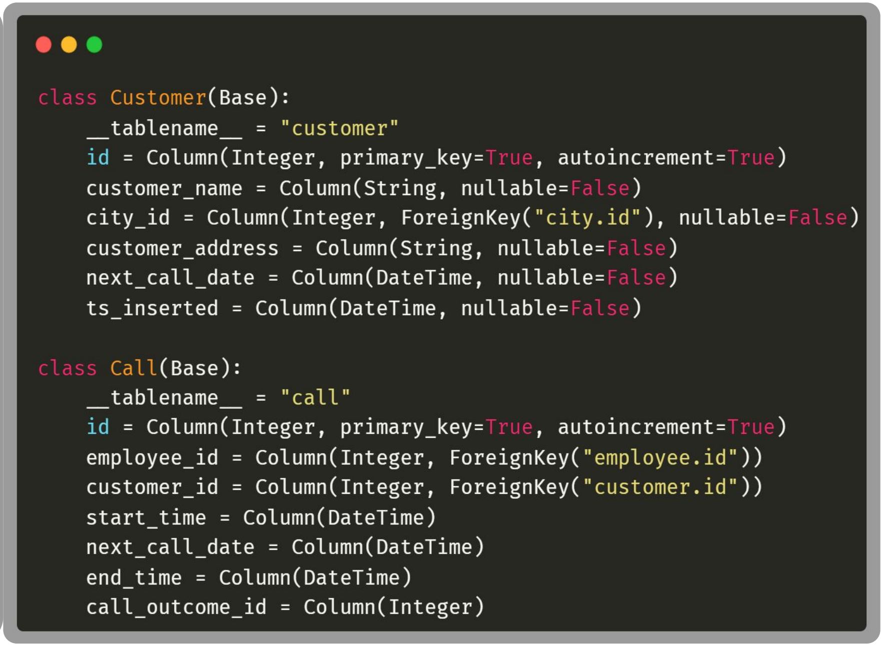
Examples: Models

- The first step is to create the implementations of the models presented in the database.
- The models will be implemented as follows:
    - Base: Imported from from sqlalchemy.ext.declarative, declarative_base. Acts as the foundation for all ORM models.
    - Table_Name Class: Maps to a table named Table_Name. This is declared by adding the attribute __tablename__ = 'table_name
    - Column_Name Attributes: Maps each attribute of the class with the ones inside the table. This is declared by declaring attributes as follows column_name = Column(Type, Constraint)
    - Foreign keys are declared as a constraint inside the Column using ForeignKey("table.column")

Examples: Models



```python
from sqlalchemy.orm import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey, DateTime

Base = declarative_base()

class Employee(Base):
    __tablename__ = "employee"
    id = Column(Integer, primary_key=True, autoincrement=True)
    first_name = Column(String, nullable=False)
    last_name = Column(String, nullable=False)

class City(Base):
    __tablename__ = "city"
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
```

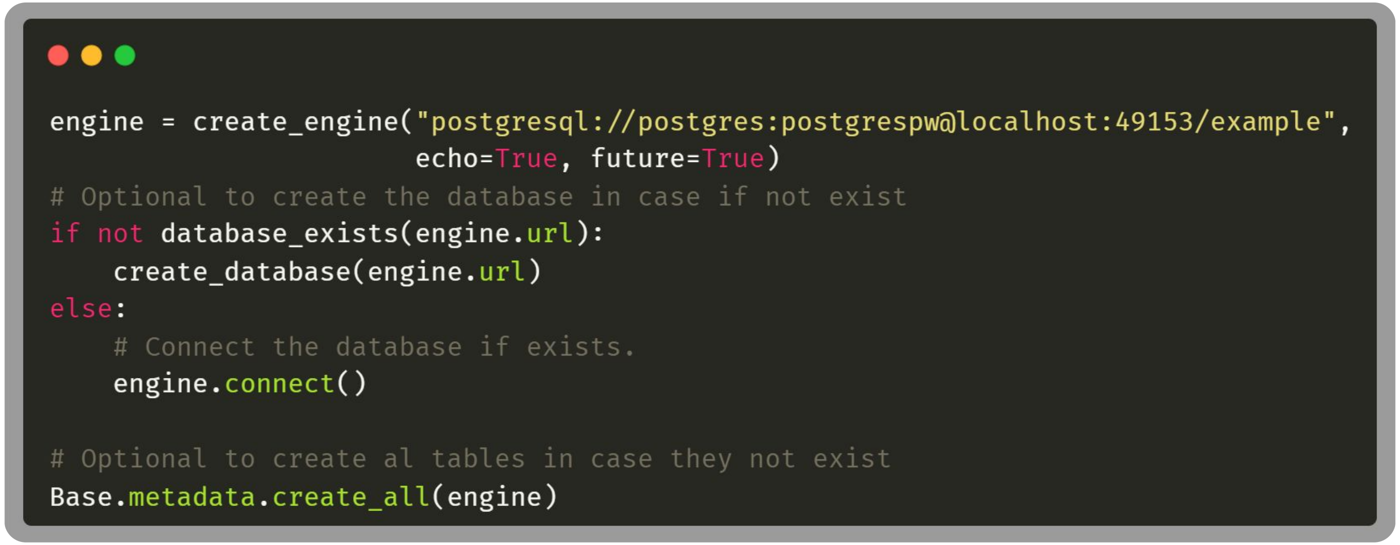


```python
class Customer(Base):
    __tablename__ = "customer"
    id = Column(Integer, primary_key=True, autoincrement=True)
    customer_name = Column(String, nullable=False)
    city_id = Column(Integer, ForeignKey("city.id"), nullable=False)
    customer_address = Column(String, nullable=False)
    next_call_date = Column(DateTime, nullable=False)
    ts_inserted = Column(DateTime, nullable=False)

class Call(Base):
    __tablename__ = "call"
    id = Column(Integer, primary_key=True, autoincrement=True)
    employee_id = Column(Integer, ForeignKey("employee.id"))
    customer_id = Column(Integer, ForeignKey("customer.id"))
    start_time = Column(DateTime)
    next_call_date = Column(DateTime)
    end_time = Column(DateTime)
    call_outcome_id = Column(Integer)
```

Examples: Connection

1. Create_engine: Creates a connection to the PostgreSQL database.
   1.1. The connection URL is specified as: postgresql://<username>:<password>@<host>:<port>/<database>
   1.2. echo=True: Logs all SQL commands sent to the database for debugging purposes.
   1.3. future=True: Opts in for newer SQLAlchemy behavior, making the API more predictable and explicit.
2. Database connection:
   2.1. database_exists(engine.url): Checks if the database specified in the engine.url exists.
   2.2. create_database(engine.url): Creates the database if it doesn't exist. Useful for automating setup during the initial run.
   2.3. engine.connect(): Establishes a connection to the existing database if it already exists.
3. Tables creation:
   3.1. Base.metadata.create_all(engine): Creates all the tables defined in the SQLAlchemy models using the Base class. If tables already exist, this does nothing, ensuring it's safe to run multiple times.

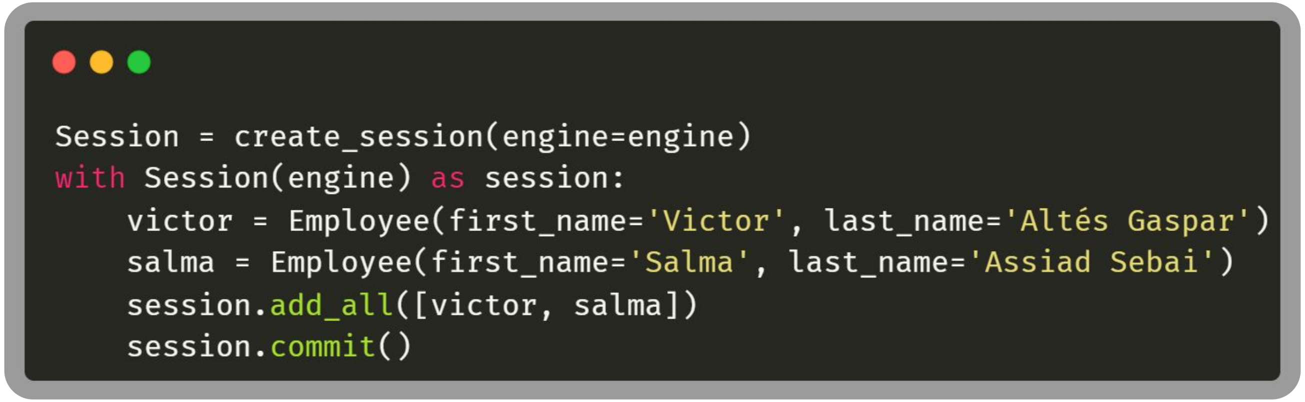
Examples: Models



```python
engine = create_engine("postgresql://postgres:postgrespw@localhost:49153/example",
                       echo=True, future=True)
# Optional to create the database in case if not exist
if not database_exists(engine.url):
    create_database(engine.url)
else:
    # Connect the database if exists.
    engine.connect()

# Optional to create al tables in case they not exist
Base.metadata.create_all(engine)
```

Examples: Insert

- To insert data in the DataBase it is only needed the instantiation of the objects belonging to one of the model classes created previously.
- Then the following SQLAlchemy functions can be used to persist the data.
  - The session can be created using from sqlalchemy.orm import sessionmaker and then calling the sessionmaker function with the engine.
  - session.add(instance): Adds a single object to the session.
  - session.add_all([instance1, instance2]): Adds multiple objects to the session.
  - session.commit(): Persists all changes to the database.



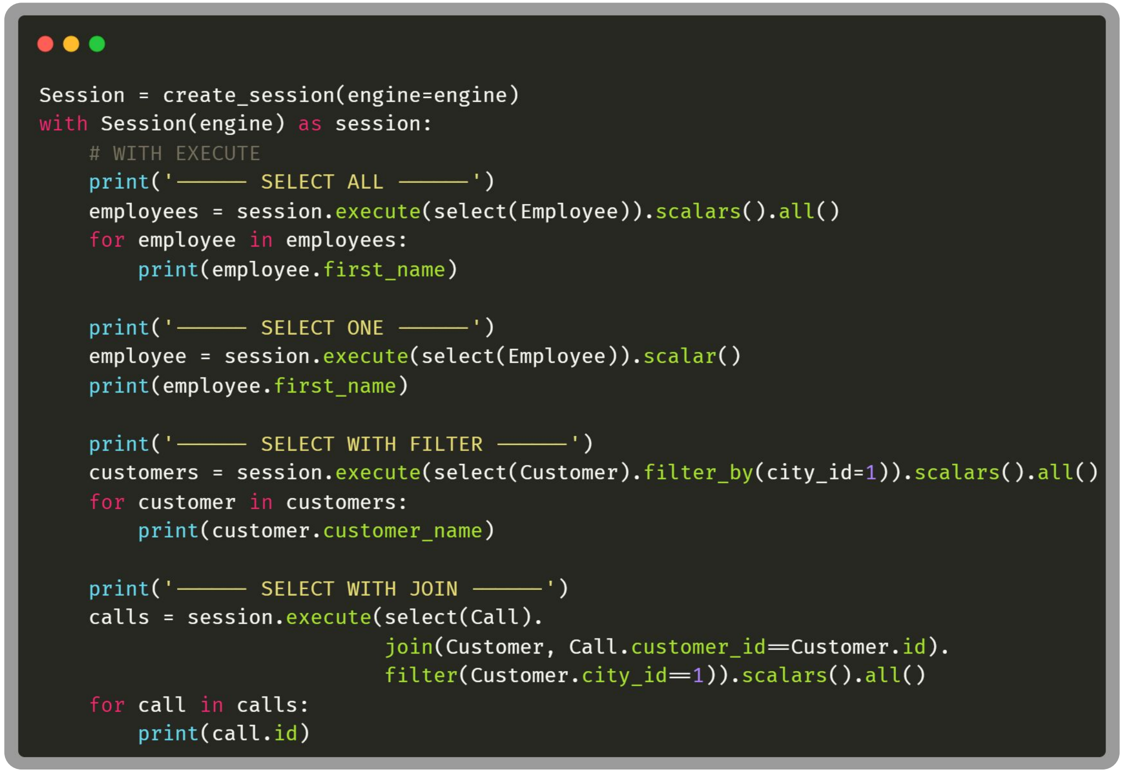
```
Session = create_session(engine=engine)
with Session(engine) as session:
    victor = Employee(first_name='Victor', last_name='Altés Gaspar')
    salma = Employee(first_name='Salma', last_name='Assiad Sebai')
    session.add_all([victor, salma])
    session.commit()
```

Examples: Select

- Selects can be done in two ways:
  - Using Execute, expects an SQL Query to be passed so it can execute the instruction, the query can be build manually or using the select with Model classes.
  - Using Query and Model classes data can be also queried.
- To add functionality over the selects there is a list of different functions that can be called after the execute or query function.
  - .all(): Fetch all results as a list.
  - .first(): Fetch the first result or None if no match.
  - .one(): Fetch exactly one result; raises if not exactly one.
  - .filter(): Apply conditions to the query (flexible).
  - .filter_by(): Apply conditions with keyword arguments (simpler).
  - .order_by(): Sort the results by one or more columns.
  - .limit(n): Limit the number of rows returned.
  - .offset(n): Skip the first n rows.
  - .scalars().all(): Fetch all results as a list (for Execute queries).
  - .scalar(): Fetch the first result (for Execute queries).
  - .scalar_one_or_none(): Fetch the first result or None if no match (for Execute queries).

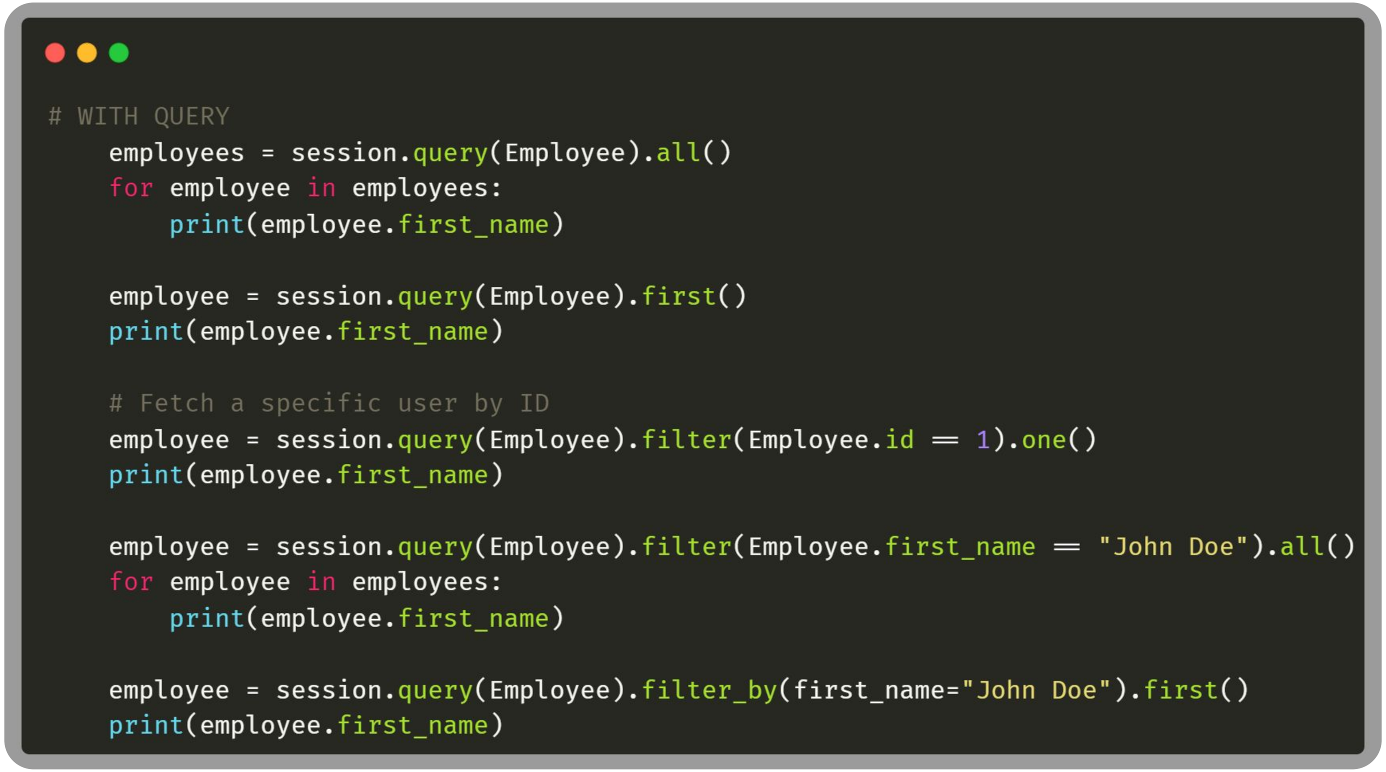
Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Examples: Select



```python
Session = create_session(engine=engine)
with Session(engine) as session:
    # WITH EXECUTE
    print('------ SELECT ALL ------')
    employees = session.execute(select(Employee)).scalars().all()
    for employee in employees:
        print(employee.first_name)

    print('------ SELECT ONE ------')
    employee = session.execute(select(Employee)).scalar()
    print(employee.first_name)

    print('------ SELECT WITH FILTER ------')
    customers = session.execute(select(Customer).filter_by(city_id=1)).scalars().all()
    for customer in customers:
        print(customer.customer_name)

    print('------ SELECT WITH JOIN ------')
    calls = session.execute(select(Call).
                                join(Customer, Call.customer_id==Customer.id).
                                filter(Customer.city_id==1)).scalars().all()
    for call in calls:
        print(call.id)
```

Examples: Select

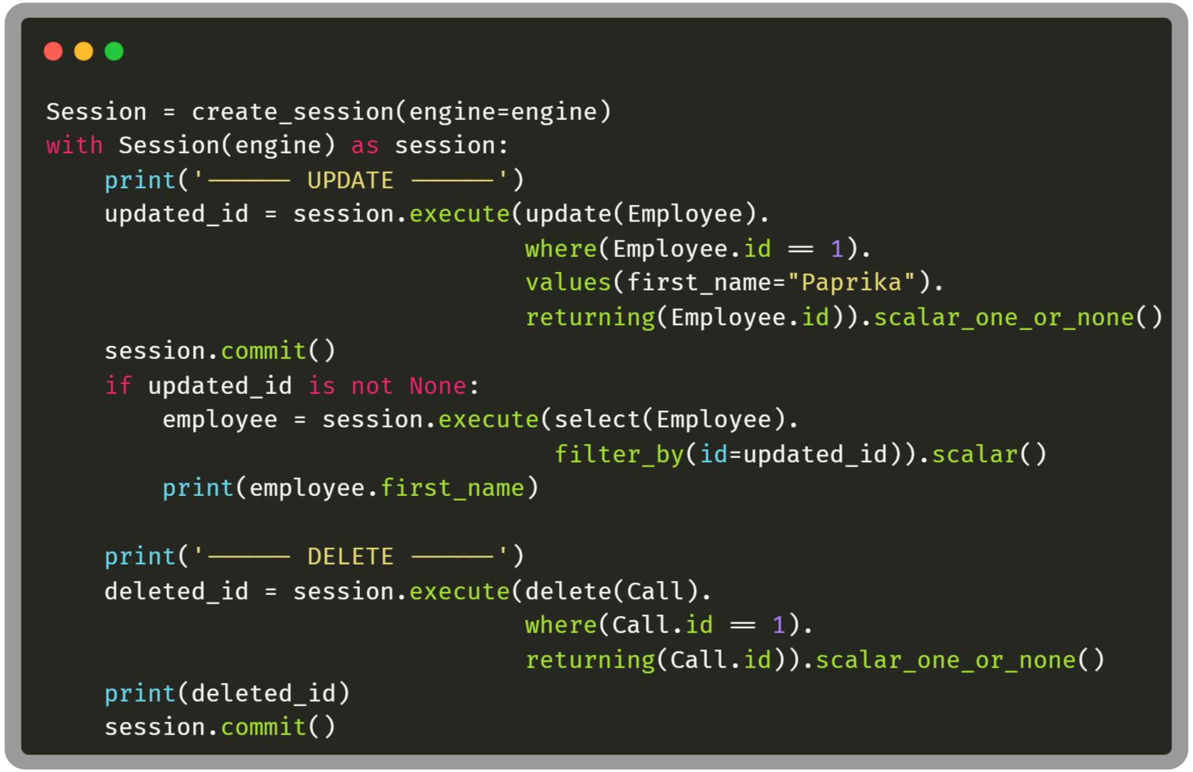


```python
# WITH QUERY
    employees = session.query(Employee).all()
    for employee in employees:
        print(employee.first_name)

    employee = session.query(Employee).first()
    print(employee.first_name)

    # Fetch a specific user by ID
    employee = session.query(Employee).filter(Employee.id == 1).one()
    print(employee.first_name)

    employee = session.query(Employee).filter(Employee.first_name == "John Doe").all()
    for employee in employees:
        print(employee.first_name)

    employee = session.query(Employee).filter_by(first_name="John Doe").first()
    print(employee.first_name)
```

Examples: Delete/Update

- To Delete and Update queries the functions update and delete are available to use by means of execute.
- The behaviour is the same as the one mentioned in the select section, by passing a query inside the delete or update function build manually or using the model classes it will delete or update the desired rows.

Examples: Delete/Updates



```python
Session = create_session(engine=engine)
with Session(engine) as session:
    print('——— UPDATE ———')
    updated_id = session.execute(update(Employee).
                                 where(Employee.id == 1).
                                 values(first_name="Paprika").
                                 returning(Employee.id)).scalar_one_or_none()
    session.commit()
    if updated_id is not None:
        employee = session.execute(select(Employee).
                                   filter_by(id=updated_id)).scalar()
        print(employee.first_name)

    print('——— DELETE ———')
    deleted_id = session.execute(delete(Call).
                                 where(Call.id == 1).
                                 returning(Call.id)).scalar_one_or_none()
    print(deleted_id)
    session.commit()
```

Examples: Link

Models: Link
Inserts: Link
Update/Delete: Link
Select: Link