ЛР8. Работа с БД из приложения

Работа с базой данных из приложения

В общем случае, работа с БД из приложения в простейшем случае состоит из следующих шагов:

- Подключение к БД;
- Открытие соединения с БД;
- Выполнение запроса(ов) к БД;
- Закрытие соединения.

Синтаксис выполнения этих операций зависит от используемого языка программирования и библиотек.

В частности, выполнение SQL-запросов к БД с использованием Python и библиотеки MySQL Connector выглядит следующим образом.

Соединение с MySQL сервером:

```
import mysql.connector

cnx = mysql.connector.connect(user='root', password='password',
host='127.0.0.1',
database='sakila')
cnx.close()
```

После завершения работы с текущим соединением необходимо его закрыть с помощью метода *close*.

Выполнение SQL запросов к БД в библиотеке MySQL Connector осуществляется с помощью класса MySQLCursor, экземпляр которого также называется курсором:

```
cursor = cnx.cursor()
```

Для непосредственно выполнения запроса используется метод `cursor.execute(operation, params=None, multi=False)` который принимает на вход текст SQL запроса *operation*, параметры запроса *params*, которые отмечаются в тексте запроса %s или %(name)s в зависимости от

способа передачи параметров (кортеж или словарь). Если параметр *multi=True,* то метод *execute* возвращает итератор и параметр *operation* может содержать несколько запросов. Ниже представлены примеры выполнения запросов.

Вставка данных:

```
from datetime import date

add_employee = ("INSERT INTO employees "
                "(first_name, last_name, birth_date) "
                "VALUES (%s, %s, %s)")
data_employee = ('Geert', 'Vanderkelen', date(1977, 6, 14))
cursor.execute(add_employee, data_employee)

cnx.commit() # по умолчанию автоматическое подтверждение транзакций в
MySQL Connector отключено
cursor.close()
cnx.close()
```

Выполнение SELECT запроса:

```
import datetime

query = ("SELECT first_name, last_name, birth_date FROM employees "
         "WHERE birth_date BETWEEN %s AND %s")

birth_start = datetime.date(1989, 1, 1)
birth_end = datetime.date(1989, 12, 31)
cursor.execute(query, (birth_start, birth_end))

for (first_name, last_name, birth_date) in cursor:
  print("{}, {} was born on {:%d %b %Y}".format(
    last_name, first_name, birth_date))

cursor.close()
cnx.close()
```

Object-Relational Mapping

Object-Relational Mapping (ORM) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Библиотеки ORM существуют для разных языков программирования. Технология ORM позволяет проектировать работу с данными в терминах классов, а не таблиц данных. Она позволяет преобразовывать классы в данные,

пригодные для хранения в базе данных, причем схему преобразования определяет сам разработчик. Кроме того, ORM предоставляет простой API-интерфейс для CRUD-операций над данными. Благодаря технологии ORM нет необходимости писать SQL-код для взаимодействия с базой данных.

ORM библиотеки включают в себя: Hibernate в Java, Entity Framework, Dapper в .Net, SQLAlchemy в Python и т.д.

Одним из наиболее популярным средством реализации технологии ORM на языке Python является модуль SQLAlchemy, подробное описание которого представлено на официальном сайте: https://docs.sqlalchemy.org/en/14/orm/tutorial.html. Ниже представлен пример работы с модулем SQLAlchemy.

Импорт необходимых модулей/классов:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from datetime import date
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, String, Integer, Date, Table,
ForeignKey, Boolean
from sqlalchemy.orm import relationship, backref
```

Подключение к MySQL серверу:

```
engine =
create_engine('mysql+mysqlconnector://username:password@127.0.0.1/db
name')
Session = sessionmaker(bind=engine)
session = Session()
```

Объявление классов:

```
Base = declarative_base()

class Musician(Base):
    __tablename__ = 'musician'

    id = Column(Integer, primary_key=True)
    name = Column(String(20))
    birthday = Column(Date)

    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
```

```
class ContactDetails(Base):
    __tablename__ = 'contact_details'

    id = Column(Integer, primary_key=True)
    phone_number = Column(String(20))
    address = Column(String(20))
    musician_id = Column(Integer, ForeignKey('musician.id'))
    musician = relationship("Musician", backref="contact_details")

    def __init__(self, phone_number, address, musician):
        self.phone_number = phone_number
        self.address = address
        self.musician = musician
```

Создание объектов классов и запись в БД:

```
ed_simons = Musician("Ed Simons", date(1970, 6, 9))
tom_rowl = Musician("Tom Rowlands", date(1971, 1, 11))
ed_contact = ContactDetails("4155552671", "London", ed_simons)
tom_contact = ContactDetails("4235555623", "London", tom_rowl)
tom_contact2 = ContactDetails("4214442323", "Manchester", tom_rowl)
session.add(ed_simons)
session.add(tom_rowl)
session.add(ed_contact)
session.add(tom_contact)
session.add(tom_contact2)
session.commit()
```

Выполнение запросов:

#Вывести имена всех музыкантов

```
musicians = session.query(Musician).all()
for musician in musicians:
    print(f'{musician.name}')
print('')
```

#Вывести всех музыкантов и их номера телефона, которые имеют несколько адресов проживания

```
query = session.query(Musician).join(ContactDetails).group_by(Musician.id).having(func.count(ContactDetails.address) > 1).all()

print('### Musicians with several houses:')
for musician in query:
    print(musician.name, [cnt.phone_number for cnt in musician.contact_details])

session.close()
```

Подготовленные запросы (Prepared statements)

https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-prepared-statements.html

Подготовленные запросы – это часть функциональности SQL-баз данных, предназначенная для отделения данных запроса и собственно выполняемого SQL-запроса.

Использование подготовленных запросов позволяет уменьшить вычислительную нагрузку на синтаксический анализ SQL запросов при каждом их выполнении. Кроме того, подготовленные запросы позволяют защититься от SQL инъекций. Например, пусть SQL запрос, сформированный в СУБД или приложении, имеет вид:

```
CONCAT("SELECT * FROM users WHERE name = ",userName,";"),
```

где `userName` – это строка, вводимая пользователем. В том случае, если пользователь введет строку `userName = " 'name' or '1'='1' "`, то он сможет получить содержимое всей таблицы `users`. Использование подготовленных запросов позволяет избежать подобных ситуаций.

Выражение `PREPARE` подготавливает SQL запрос и присваивает ему имя `stmt_name`:

```
PREPARE stmt_name FROM preparable_stmt
```

Здесь `preparable_stmt` – это строковая переменная, которая содержит текст SQL запроса. Текст должен представлять только один запрос. В пределах данного запроса символы «?» могут использоваться в качестве маркеров параметров для указания места использования данных в запросе. Символы «?» не должны быть заключены в кавычки. Маркеры параметров могут использоваться только там, где должны находиться данные, а не для ключевых слов SQL, идентификаторов и т.д.

После подготовки SQL запроса с помощью `PREPARE`, оно выполняется с помощью команды `EXECUTE`:

```
EXECUTE stmt_name [USING @var_name [, @var_name] ...]
```

Если подготовленный запрос содержит маркеры параметров, необходимо использовать выражение `USING`, в котором перечисляются пользовательские переменные, содержащие связанные с параметрами значения. Значения параметров могут быть представлены только пользовательскими переменными, в выражении

USING должно быть указано такое количество переменных, которое соответствует числу маркеров параметров «?» в SQL запросе.

После завершения работы с подготовленным запросом выполняется команда:

```
DEALLOCATE PREPARE stmt_name
```

При работе с базой данных из приложения также необходимо использовать подготовленные запросы, чтобы исключить возможность выполнения SQL_инъекции. Синтаксис создания подготовленных запросов зависит от используемого языка программирования и библиотеки. Например, выполнение подготовленного запроса с использованием Python может выглядеть следующим образом:

```
cnx = mysql.connector.connect(…)  # создание подключения
cursor = cnx.cursor()  # открытие соединения
sql = cursor.execute("SELECT id FROM country_list where
country_name=%s", (target,))  # выполнение запроса
```

В данном примере target является параметром запроса, %s – маркером параметра.