Урок 4

Работа с базами данных. SQLAlchemy

Python DB-API.

Подключение к базе данных, объект курсора, выполнение SQL-запросов.

Использование ORM для работы с базами данных.

ORM SQLAlchemy. Классический и декларативный стиль работы. Объект сессии.

Введение

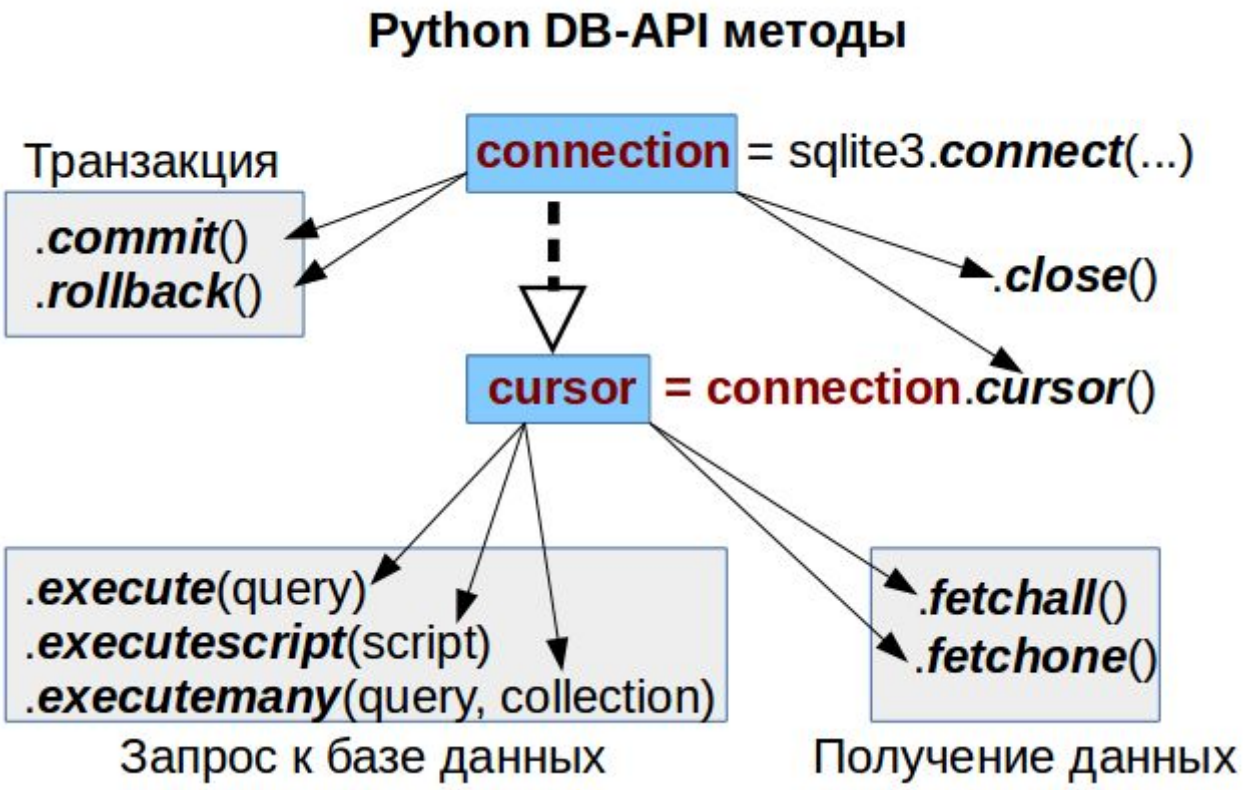
Данный урок посвящён изучению взаимодействия Python-кода с реляционными базами данных. Будут рассмотрены два подхода: с использованием Python DB-API и с использование ORM-библиотеки SQLAlchemy.

Здесь подразумевается, что слушатель имеет базовые представления о реляционных базах данных (таблицы, отношения, ключи, индексы) и языке SQL (добавление, редактирование, выборка данных), т.к. формат курса и урока не позволяют охватить всю возможную информацию и ориентирован в первую очередь на изучение возможностей особенностей языка Python и его библиотек.

Большая часть примеров урока подразумевает использование реляционной СУБД SQLite, т.к. она не требует установки дополнительного ПО, а модуль sqlite3 является частью стандартной библиотеки Python. Там, где это необходимо, приводятся примеры взаимодействия с другими СУБД.

Python DB-API

Python DB-API – это не конкретная библиотека, а набор правил, которым подчиняются отдельные модули, реализующие работу с конкретными базами данных. Отдельные нюансы реализации для разных баз могут отличаться, но общие принципы позволяют использовать один и тот же подход при работе с разными базами данных.



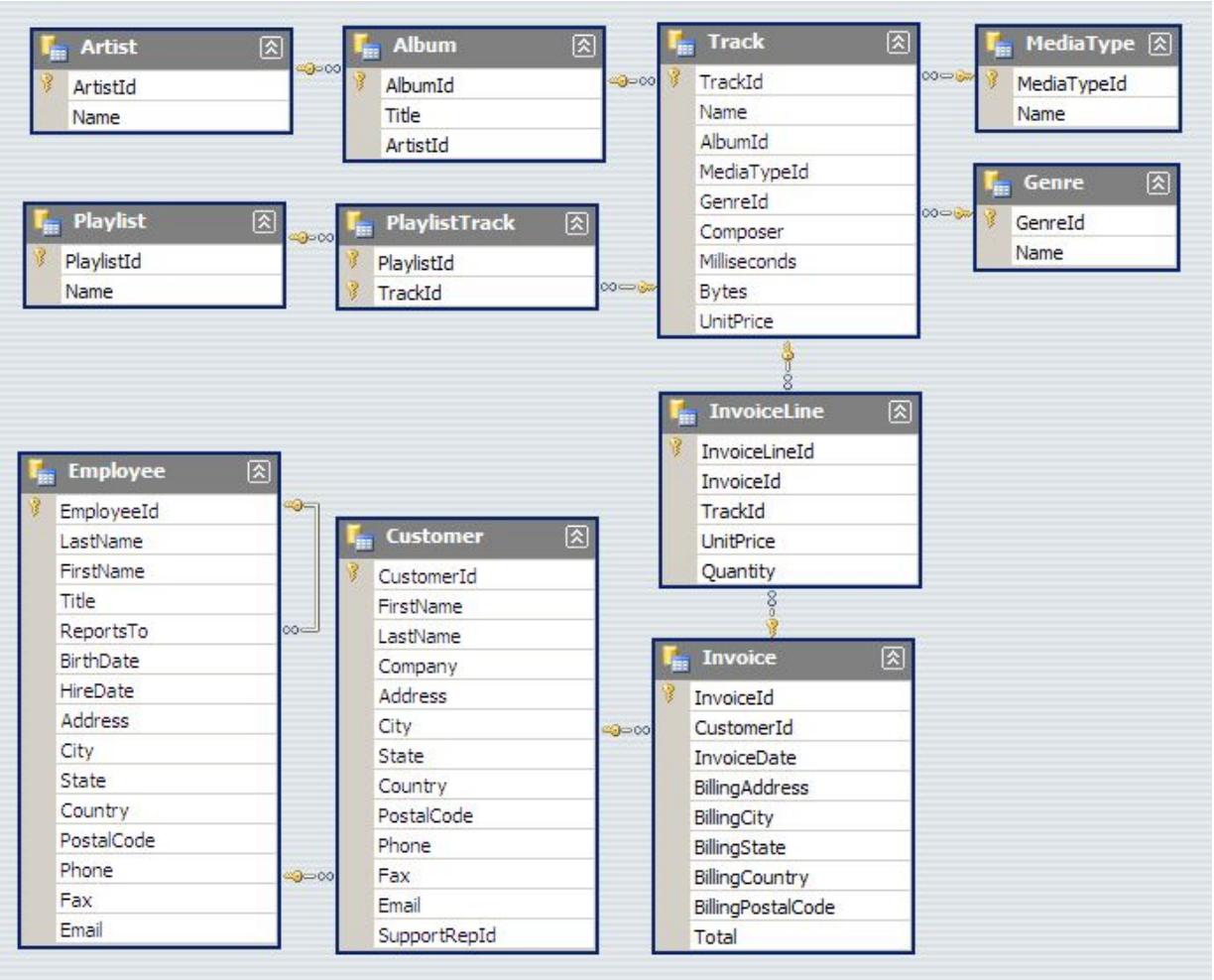
Подробное описание Python DB-API находится в документе **PEP-249** (PEP 249 – Python Database API Specification v2.0).

Необходимые инструменты

- Python имеет встроенную поддержку SQLite базы данных, для этого не требуется ничего дополнительно устанавливать, достаточно в скрипте указать импорт стандартной библиотеки:

```
import sqlite3
```

- В некоторых примерах будет использоваться тестовая база данных Chinook Database (лицензия MIT). Для примеров потребуется бинарный файл “**Chinook_Sqlite.sqlite**” (прилагается к уроку). Структура Chinook Database представлена на рисунке:



- Для удобства работы с базой (просмотр, редактирование) можно использовать программу-браузер баз данных, поддерживающую SQLite. В уроке работа с браузером не рассматривается, но он поможет Вам наглядно видеть что происходит с базой в процессе экспериментов.

Примечание: внося изменения в базу, не забудьте их применить, так как база с непримененными изменениями остается заблокированной.

Некоторые варианты браузеров БД:

- привычная вам утилита для работы с базой в составе вашей IDE;
- SQLite Database Browser;
- SQLiteStudio;

- Valentina Studio.

Библиотеки для работы с БД

Поскольку в стандартной библиотеке Python есть только модуль для взаимодействия с SQLite, модули для других СУБД нужно устанавливать дополнительно (в большинстве случаев все необходимые модули устанавливаются через pip).

Перечислим библиотеки, обеспечивающие взаимодействие с различными СУБД, которые реализуют Python DB-API:

- **SQLite**: стандартный модуль sqlite3;
- **PostgreSQL**: psycopg2, PyGreSQL;
- **MySQL**: PyMySQL, mysql-connector-python;
- **MSSQL Server**: pyodbc, pymssql.

Соединение с базой, получение курсора

Рассмотрим базовый шаблон работы с DB-API, который будет использоваться во всех дальнейших примерах:

```
# Подключение библиотеки, соответствующей типу требуемой базы данных
import sqlite3

# Создание соединения с базой данных
# В данном случае - это просто файл базы
conn = sqlite3.connect('Chinook_Sqlite.sqlite')

# Создаем курсор - это специальный объект который делает запросы и получает их результаты
cursor = conn.cursor()

# ========== ТУТ БУДЕТ КОД РАБОТЫ С БАЗОЙ ДАННЫХ ===========
# ===== КОД ДАЛЬНЕЙШИХ ПРИМЕРОВ ВСТАВЛЯТЬ В ЭТО МЕСТО =============

# В конце необходимо закрыть соединение с базой данных
conn.close()
```

При работе с другими базами данных могут использоваться дополнительные параметры соединения, например, для PostrgeSQL:

```
conn = psycopg2.connect(host=hostname, user=username, password=password, dbname=database)
```

Чтение из базы

Для получения данных из БД нужно выполнить SQL-запрос через метод курсора `execute()`, после чего получить данные один из fetch-методов:

```
# Выполняется SELECT запрос к базе данных, используя обычный SQL-синтаксис
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")

# Получение результатов сделанного запроса
results = cursor.fetchall()
results2 =  cursor.fetchall()

print(results)
# [('A Cor Do Som',), ('Aaron Copland & London Symphony Orchestra',), ('Aaron Goldberg',)]
print(results2)
# []
```

Обратите внимание: после получения результата из курсора, второй раз без повторения самого запроса его получить нельзя — вернется пустой результат!

Длинные запросы можно разбивать на несколько строк в произвольном порядке, если они заключены в тройные кавычки — одинарные (‘‘‘…’‘‘) или двойные (“””...”””):

```
cursor.execute("""
  SELECT name
  FROM Artist
  ORDER BY Name LIMIT 3
""")
```

Запись в базу

Добавление записей в БД осуществляется путём выполнения (**execute**) SQL-запроса `INSERT` и последующего подтверждения транзакции (**commit**):

```
# Выполняется INSERT запрос к базе данных, используя обычный SQL-синтаксис
cursor.execute("insert into Artist values (Null, 'A Aagrh!') ")

# Если выполняются изменения в базе данных - необходимо сохранить транзакцию
conn.commit()
# Проверка результатов
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")
results = cursor.fetchall()
print(results)  # [('A Aagrh!',), ('A Cor Do Som',), ('Aaron Copland & London Symphony Orchestra',)]
```

Примечание: если к базе установлено несколько соединений и одно из них осуществляет модификацию базы, то база SQLite блокируется до завершения (метод соединения .commit()) или отмены (метод соединения .rollback()) транзакции.

Несколько запросов за один раз

Метод курсора `.execute()` позволяет делать только один запрос за раз, при попытке сделать несколько через точку с запятой будет ошибка:

```
cursor.execute("""
  insert into Artist values (Null, 'A Aagrh!');
  insert into Artist values (Null, 'A Aagrh-2!');
""")
# Будет получена ошибка
# sqlite3.Warning: You can only execute one statement at a time.
```

Для решения такой задачи можно либо несколько раз вызвать метод курсора `.execute()`:

```
cursor.execute("""insert into Artist values (Null, 'A Aagrh!');""")
cursor.execute("""insert into Artist values (Null, 'A Aagrh-2!');""")
```

Либо использовать метод курсора `.executescript()`:

```
cursor.executescript("""
 insert into Artist values (Null, 'A Aagrh!');
 insert into Artist values (Null, 'A Aagrh-2!');
""")
```

Данный метод удобен, когда запросы сохранены в отдельной переменной или файле и требуется применить такой запрос к базе данных.

Подстановка значений в запрос

- **Важно!** Никогда, ни при каких условиях, не используйте конкатенацию строк (+) или форматную строку для передачи переменных в SQL-запрос. Такое формирование запроса, при возможности попадания в него пользовательских данных – это место для **SQL-инъекций**!

Правильный способ – использование второго аргумента метода .execute()

В SQLite возможны два варианта:

```
# 1. С подстановкой по порядку на места знаков вопросов:
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT ?", ('2'))
# 2. С использованием именованных замен:
cursor.execute("SELECT Name from Artist ORDER BY Name LIMIT :limit", {"limit": 3})
```

Параметр paramstyle определяет какой именно стиль используется для подстановки переменных в данном модуле:

```
import sqlite3
paramstyle = sqlite3.paramstyle

if paramstyle == 'qmark':
    ph = "?"
elif paramstyle == 'format':
    ph = "%s"
else:
    raise Exception("Unexpected paramstyle: %s" % paramstyle)

sql = "INSERT INTO foo VALUES (%(ph)s, %(ph)s, %(ph)s)" % { "ph" : ph }
```

Множественная подстановка значений

Для подстановки списка значений в запрос необходимо использовать метод курсора .executemany():

```
# Обратите внимание, даже передавая одно значение - его нужно передавать кортежем!
# Именно поэтому тут используется запятая в скобках!
new_artists = [
    ('A Aagrh!',),
    ('A Aagrh!-2',),
    ('A Aagrh!-3',),
]
cursor.executemany("insert into Artist values (Null, ?);", new_artists)
```

Получение результатов

Для получения данных выборки (SELECT-запрос) могут применяться методы курсора:

- fetchone() - возвращает одну строку результата запроса, повторный вызов получает следующую строку и т.д. Всегда возвращает кортеж или None, если запрос пустой;
- fetchmany([size=cursor.arraysize]) - возвращает набор строк результата, указанного размера;
- fetchall() - получает все строки результата запроса.

Пример получения данных, используя метод курсора .fetchone():

```
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")
print(cursor.fetchone())    # ('A Cor Do Som',)
print(cursor.fetchone())    # ('Aaron Copland & London Symphony Orchestra',)
print(cursor.fetchone())    # ('Aaron Goldberg',)
print(cursor.fetchone())    # None
```

Важно! Стандартный курсор забирает все данные с сервера сразу, вне зависимости от использования .fetchall() или .fetchone().

Курсор как итератор

Для удобства можно использовать объект курсора в качестве итератора:

```
# Использование курсора как итератора
for row in cursor.execute('SELECT Name from Artist ORDER BY Name LIMIT 3'):
        print(row)

# Полученный результат:
# ('A Cor Do Som',)
# ('Aaron Copland & London Symphony Orchestra',)
# ('Aaron Goldberg',)
```

Обработка ошибок

Для большей устойчивости программы (особенно при операциях записи) следует оборачивать инструкции обращения к БД в блоки try-except-else и использовать встроенный в sqlite3 “родной” объект ошибок, например, так:

```
try:
    cursor.execute(sql_statement)
    result = cursor.fetchall()
except sqlite3.DatabaseError as err:
    print("Error: ", err)
else:
    conn.commit()
```

Использование оператора with

Некоторые библиотеки для взаимодействия с БД предоставляют интерфейс менеджера контекста (методы __enter__ и __exit__) для своих классов, что позволяет безопасно взаимодействовать с объектами этих классов (напомним, что освобождение ресурсов, закрытие соединений и некоторые другие сервисные действия в этом случае возлагаются на менеджер контекста - см. тему “Менеджеры контекста” данного курса).

В частности, интерфейс менеджера контекста имеют объекты соединений (connect) и курсоров (cursor).

Пример того, как это может быть выполнено для библиотеки psycopg2 (для других библиотек в коде изменится только имя самой библиотеки):

```
# Менеджер контекста в библиотеке psycopg2
import psycopg2

with psycopg2.connect("dbname='habr'") as conn:
    with conn.cursor() as cur:
        # -------------------------
        # Код взаимодействия с БД...
        # -------------------------
```

Использование row_factory

Использование атрибута row_factory позволяет производить дополнительную обработку результата выборки (имеется доступ к метаданным запроса). По сути, row_factory - callback-функция для обработки данных при возврате строки.

Например, можно реализовать обращение к результату запроса по имени столбца. Для этого нужно воспользоваться атрибутом курсора .description. Данный атрибут возвращает сведения о столбцах для последней выборки (для каждого столбца данные представлены кортежем из 7 элементов).

Пример из документации:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

Справка по функциям Python DB-API

Модуль

- connect() - установка соединения с БД, создание объекта класса Connection;
- threadsafety - константа, указывающая уровень потокобезопасности модуля;
- paramstyle - строковая константа, задающая формат маркера подстановки данных в запрос.

Исключения (Exceptions)

- Warning - исключение, создаваемое для важных предупреждений (warnings);
- Error - базовое класс для исключений типа error;
- InterfaceError - ошибки, свойственные интерфейсу БД;
- DatabaseError - исключения свойственные базе данных;
- DataError - исключения обработки данных (деление на ноль, выход за границы диапазона);
- OperationalError - исключения, относящиеся к операциям БД, которые не подконтрольны программисту (неожиданное отключение БД, неизвестное имя источника данных, невозможность выполнить транзакцию, ошибка выделения памяти).
- IntegrityError - исключение, создаваемое при нарушении целостности отношений (неудачная проверка внешнего ключа);
- InternalError - внутреннее исключение БД (некорректный курсор, ошибка синхронизации транзакции и прочее);
- ProgrammingError - программные ошибки (таблица не найдена или уже присутствует, ошибка SQL-синтаксиса, неверное количество параметров)
- NotSupportedError - исключение создаётся при использовании метода или API, который не поддерживается базой данных.

Соединение (Connection)

- con.close() – закрывает соединение с сервером базы данных;
- con.commit() – подтверждает все незавершенные транзакции;
- con.rollback() – откатывает все изменение в базе данных до момента, когда были запущены незавершённые транзакции.

- con.cursor() – создаёт новый курсор (экземпляр класса Cursor).

Курсор (Cursor)

- cur.description - последовательность кортежей с информацией о каждом столбце в текущем наборе данных. Кортеж имеет вид (name, type_code, display_size, internal_size, precision, scale, null_ok);
- cur.rowcount - число строк, на которые повлиял последний запрос;
- c.arraysize – целое число, которое используется методом cur.fetchmany как значение по умолчанию;
- c.close() – закрывает курсор предотвращая выполнения каких либо запросов с его помощью;
- c.callproc(procname [, param]) – вызывает хранимую процедуру;
- c.execute(query [, param]) – выполняет запрос к базе данных (query);
- c.executemany(query [, paramsequence]) – многократное выполнение запросов к базе данных (query);
- c.fetchone() – возвращает следующую запись из набора данных, полученного вызовом c.execute*();
- c.fetchmany([size]) – возвращает последовательность записей из набора данных;
- c.fetchall() – возвращает последовательность всех записей оставшихся в полученном наборе данных;
- c.nextset() – пропускает все оставшиеся записи в текущем наборе данных и переходит к следующему набору;
- c.setinputsize(sizes) – сообщает курсору о параметрах, которые будут переданы в последующих вызовах методов cur.execute*();
- c.setoutputsize(sizes [, column]) – устанавливает размер буфера для определённого столбца в возвращаемом наборе данных.

Типы данных и их конструкторы

- Date(year, month, day) - формирует объект, содержащий дату;
- Time(hour, minute, second) - формирует объект, содержащий время;
- Timestamp(year, month, day, hour, minute, second) - формирует объект, содержащий временную метку;
- DateFromTicks(ticks) - формирует объект-дату из количества секунд;
- TimeFromTicks(ticks) - формирует объект-время из количества секунд;
- TimestampFromTicks(ticks) - формирует объект дата-время из количества секунд;
- Binary(string) - формирует объект с бинарными данными;
- STRING - тип для представления строковых столбцов таблицы (CHAR);
- BINARY - тип для представления бинарных столбцов таблицы (LONG, RAW, BLOB);
- NUMBER - тип для представления числовых столбцов таблицы;
- DATETIME - тип для представления столбцов дата-время;
- ROWID - тип для представления “Row ID”-столбцов;
- **NULL-значения** представляются Python-объектом None как при вводе, так и при выводе.

SQLAlchemy

SQLAlchemy — это программная библиотека на языке Python для работы с реляционными СУБД с применением технологии ORM (англ. **Object-Relational Mapping**, рус. объектно-реляционное отображение - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая “виртуальную объектную базу данных”). Служит для синхронизации объектов Python и записей реляционной базы данных. SQLAlchemy позволяет описывать структуры баз данных и способы взаимодействия с ними на языке

Python без использования SQL. Библиотека была выпущена в феврале 2006 под лицензией открытого ПО MIT.

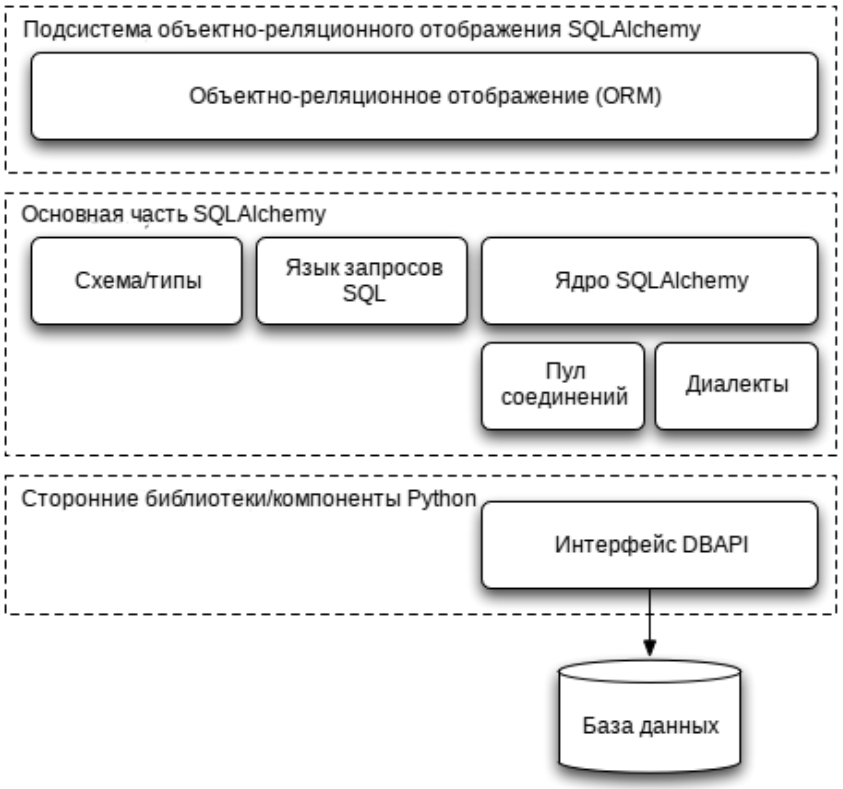
Работает back-end для баз данных: MySQL, PostgreSQL, SQLite, Oracle и других, между которыми можно переключаться простым изменением конфигурации.

Преимущества использования

Использование SQLAlchemy для автоматической генерации SQL-кода имеет несколько преимуществ по сравнению с ручным написанием SQL-запросов:

- **Безопасность**. Параметры запросов экранируются, что делает атаки типа внедрение SQL-кода маловероятными.
- **Производительность**. Повышается вероятность повторного использования запроса к серверу базы данных, что может позволить ему в некоторых случаях применить повторно план выполнения запроса.
- **Переносимость**. SQLAlchemy, при должном подходе, позволяет писать код на Python, совместимый с несколькими back-end СУБД. Несмотря на стандартизацию языка SQL, между базами данных имеются различия в его реализации, абстрагироваться от которых и помогает SQLAlchemy.

Архитектура SQLAlchemy



SQLAlchemy предоставляет богатый API для каждого уровня взаимодействия с БД, разбивая общую задачу взаимодействия на 2 категории: **ядро (Core)** и **объектно-реляционное представление (ORM)**. Ядро включает в себя взаимодействие Python DB-API, обработку текстовых SQL-запросов и управление схемой БД (все эти части предоставляют API). ORM-часть - это библиотека построенная поверх ядра SQLAlchemy (любой разработчик может создать свою ORM-библиотеку поверх ядра SQLAlchemy).

Разделение Ядро/ORM всегда было отличительной особенностью SQLAlchemy (есть как плюсы, так и минусы). Ядро SQLAlchemy позволяет ORM-слою:

- связывать в структуру под названием Table атрибуты Python-класса, а не имена полей из БД;
- для формирования SELECT-запроса использовать структуру под названием select, а не формировать строку запроса из разных частей;
- получать результат запроса через “фасад” (шаблон программирования) под названием ResultProxy, который отображает select-структуру на каждую строку результата, а не передавать данные из курсора БД в пользовательские объекты.

Элементы ядра могут быть не видны в самом простом ORM-приложении. Однако в виду того, что Ядро аккуратно встроено в ORM (для обеспечения плавного перехода между конструкциями ORM и Ядра), более сложное ORM-приложение может пропустить один или два уровня абстракции для того, чтобы взаимодействовать с БД, используя специфические и улучшенные настройки (если требуется).

Обратной стороной подхода “ORM/Ядро” является то, что инструкции должны пройти большее количество этапов. Стандартная реализация CPython имеет особенность вызова Python-функций, которая снижает быстродействие. Для обхода этой ситуации можно сокращать цепочки вызовов функций и переносить критичные к быстродействию участки на язык C. Разработчики SQLAlchemy используют оба подхода для улучшения производительности (интерпретатор PyPy позволяет обходится без трюков с улучшением быстродействия).

Установка SQLAlchemy

Установка SQLAlchemy стандартна:

```
pip install SQLAlchemy
```

Также можно просто скачать архив с SQLAlchemy с официального сайта и выполнить установочный скрипт setup.py:

```
python setup.py install
```

Для проверки правильности установки следует проверить версию библиотеки:

```
import sqlalchemy
print("Версия SQLAlchemy:", sqlalchemy.__version__) # посмотреть версию SQLALchemy
```

Объектно-реляционная модель SQLAlchemy

Соединение с базой данных

Для упрощения демонстрации работы с SQLAlchemy также будет использоваться БД SQLite, хранящаяся в памяти.

Для соединения с СУБД используется функция create_engine():

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///:memory:', echo=True)
```

Флаг echo включает ведение лога через стандартный модуль `logging` Питона. Когда он включен, будут отображаться все создаваемые SQL-запросы.

По умолчанию соединение с БД через 8 часов простоя обрывается. Чтобы это не случилось нужно добавить опцию:

```
pool_recycle = 7200
```

и тогда каждые два часа соединение будет переустанавливаться.

Примеры создания подключений к базам данных PostgreSQL и MySQL:

```
# Создание подключения к локальной базе данных PostgreSQL
from sqlalchemy import create_engine
engine                                                                    =
create_engine('postgresql+psycopg2://username:password@localhost:5432/mydb')

# Создание подключения к удаленной базе данных MySQL
from sqlalchemy import create_engine
engine                                                                    =
create_engine('mysql+pymysql://cookiemonster:chocolatechip@mysql01.monster.intern
al'/
              '/cookies', pool_recycle=3600)
```

В рамках SQLAlchemy реализован фасадный класс для классического взаимодействия с DB-API. Точкой входа этого фасадного класса является вызов create_engine, с помощью которого устанавливается соединение и собирается конфигурационная информация. В качестве результата выполнения вызова возвращается экземпляр класса Engine. Этот объект представляет только способ осуществления запроса через DB-API, причем последний никогда непосредственно не раскрывается.

Для простого выполнения запросов объект Engine предоставляет интерфейс, известный под названием “интерфейс явного исполнения запросов” (“implicit execution interface”). Работа по созданию и закрытию соединения с базой данных и курсора посредством DB-API выполняется незаметно для разработчика:

```
engine = create_engine("postgresql://user:pw@host/dbname")
result = engine.execute("select * from table")
print(result.fetchall())
```

Создание таблиц

Далее необходимо “рассказать” SQLAlchemy о таблицах в базе данных.

Рассмотрим пример одиночной таблицы `users`, в которой хранятся записи о конечных пользователях, которые посещают некий сайт N. Необходимо определить таблицу внутри каталога `MetaData`, используя конструктор `Table()`, который похож на SQL-запрос `CREATE TABLE`:

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()
users_table = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('fullname', String),
    Column('password', String)
)
```

Далее необходимо выполнить запрос CREATE TABLE, параметры которого будут взяты из метаданных нашей таблицы. Для этого вызывается метод create_all() с параметром engine, который указывает на базу. При выполнении метода автоматически будет проверено присутствие такой таблицы перед ее созданием, так что можно выполнять этот метод много раз - ничего страшного не случится:

```
metadata.create_all(engine)
```

- **Обратите внимание**: колонки VARCHAR создаются без указания их длины - для SQLite это вполне допустимый тип данных, но во многих других СУБД так делать нельзя. Для того, чтобы выполнить этот урок в PostgreSQL или MySQL, длина должна быть определена для строк:

```
Column('name', String(50))
```

Поле “длина” в строках `String`, как и простая разрядность/точность в `Integer`, `Numeric` и т.п. не используются более нигде, кроме как при создании таблиц.

Определение класса Python для отображения в таблицу

В то время, как класс Table хранит информацию о нашей БД, он ничего не говорит о логике объектов, что используются нашим приложением. SQLAlchemy считает это отдельной задачей. Для соответствия нашей таблице users создадим элементарный класс User (то есть будет совершенно новый класс):

```
class User:
    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password

    def __repr__(self):
            return "<User('%s','%s', '%s')>" % (self.name, self.fullname,
self.password)
```

Методы __init__ и __repr__ (вызывается в функции print) определены здесь для удобства. Они не обязательны и могут иметь любую форму. SQLAlchemy не вызывает __init__ напрямую.

Настройка отображения

Теперь необходимо выполнить связывание таблицы users и класса User. Эту задачу решает пакет SQLAlchemy ORM.

Для создания отображения между таблицей и классом необходимо использовать функцию mapper:

```
from sqlalchemy.orm import mapper
print(mapper(User, users_table))      #  <Mapper at 0x...; User>
```

Функция mapper() создаст новый Mapper-объект и сохранит его для дальнейшего применения, ассоциирующегося с нашим классом. Теперь создадим и проверим объект класса User:

```
from sqlalchemy.orm import mapper             # Mapper находится в пакете с ОРМ
mapper(User, users_table)                     # Создание отображения
user = User("Вася", "Василий", "qweasdzxc")
print(user)                                           # <User('Вася', 'Василий',
'qweasdzxc'>
print(user.id)                                # None
```

Атрибут id, который не определен в __init__, все равно существует из-за того, что колонка id существует в объекте таблицы users_table.

Стандартно mapper() создает атрибуты класса для всех колонок, что есть в Table. Эти атрибуты представляют собой объекты-дескрипторы и определяют функциональность класса. Она может быть очень богатой, может включать в себя возможность отслеживать изменения и АВТОМАТИЧЕСКИ подгружать данные в базу, когда это необходимо.

Поскольку SQLAlchemy не получила задание сохранить “Василия” в базу, его id имеет значение None. Когда позже будет выполнено сохранение, в этом атрибуте будет храниться некое автоматически сформированное значение.

Декларативное создание таблицы, класса и отображения

Предыдущее приближение к конфигурированию, включающее таблицу Table, пользовательский класс и вызов mapper() иллюстрируют классический пример использования SQLAlchemy (в которой очень ценится разделение задач). Большое число приложений, однако, не требуют такого разделения, и для них SQLAlchemy предоставляет альтернативный, более лаконичный стиль - **декларативный.**

```
Base = declarative_base()
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password
    def __repr__(self):
                return  "<User('%s','%s',  '%s')>"  %  (self.name,  self.fullname,
self.password)
```

Функция `declarative_base()` определяет новый класс (`Base`), от которого будут унаследованы все необходимые ORM-классы.

- **Обратите внимание**: объекты Column определены без указания строки имени - она будет получена из имени своего атрибута.

Объект Table доступен через атрибут __table__:

```
users_table = User.__table__
```

Имеющиеся метаданные `MetaData` также доступны:

```
metadata = Base.metadata
```

Создание сессии

Теперь всё готово, чтобы начать общение с базой данных. Доступ к базе данных осуществляется через механизм сессии Session. При запуске приложения, необходимо на одном уровне с create_engine() определить класс Session, который будет служить фабрикой объектов сессий (Session):

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
```

В случае, если приложение не имеет Engine-объекта базы данных, можно создать сессию:

```
Session = sessionmaker()
```

Позже, когда будет создано подключение к базе данных с помощью create_engine(), необходимо соединить его с сессией, используя configure():

```
Session.configure(bind=engine)
```

Класс Session будет создавать Session-объекты, которые привязаны к базе данных. Другие транзакционные параметры тоже можно определить вызовом функции sessionmaker().

Когда необходимо общение с базой, необходимо создать объект класса Session:

```
session = Session()
```

Сессия здесь ассоциирована с SQLite, но у нее еще нет открытых соединений с этой базой. При первом использовании она получает соединение из набора соединений, который поддерживается engine и удерживает его до тех пор, пока не будут применены все изменения и/или не будет закрыт объект сессии.

Добавление новых объектов

Для сохранения объекта User, нужно добавить его к имеющейся сессии, вызвав метод сессии add():

```
admin_user = User("vasia", "Vasiliy Pypkin", "vasia2000")
session.add(admin_user)
```

Этот объект будет находиться в ожидании сохранения, никакого SQL-запроса пока выполнено не будет. Сессия пошлет SQL-запрос, чтобы сохранить данные пользователя, как только это понадобится, используя процесс сброса на диск (flush). Если мы запросим Васю из базы, то сначала вся ожидающая информация будет сброшена в базу, а запрос последует потом.

Для примера создадим новый объект запроса (Query), который загружает User-объекты. Запрос фильтруется по атрибуту “имя=Вася”, и из результата запроса методом first() извлекается только первый результат. Возвращается тот User, который был добавлен ранее:

```
q_user = session.query(User).filter_by(name="vasia").first()
print(q_user)              # <User('vasia','Vasiliy Pypkin', 'vasia2000')>
```

На самом деле сессия определила, что та запись из таблицы, что она вернула, та же самая, что и запись, что она уже представляла в своей внутренней хэш-таблице объектов. Поэтому в результате был получен точно тот же самый объект, что и добавленный. Концепция ORM, которая работает здесь, известна как карта идентичности, обеспечивает возможность для всех операций над конкретной записью внутри сессии оперировать одним и тем же набором данных. Как только объект с неким первичным ключом появится в сессии, все SQL-запросы на этой сессии вернут тот же самый Python-объект для этого самого первичного ключа. В случае попытки поместить в эту сессию другой, уже сохраненный объект с тем же первичным ключом, будет выдана ошибка. Для добавления нескольких User-объектов необходимо использовать метод add_all():

```
# Добавить сразу несколько записей
session.add_all([User("kolia", "Cool Kolian[S.A.]","kolia$$$"),
                 User("zina", "Zina Korzina", "zk18")])
```

При изменении данных объекта, находящегося в сессии, сессия будет “знать”, что объект был изменён:

```
admin_user.password = "-=VP2001=-"
print(session.dirty)            #  IdentitySet([<User('vasia','Vasiliy  Pypkin',
'-=VP2001=-')>])
```

Атрибут сессии new хранит объекты, ожидающие сохранения в базу данных:

```
print(session.new)
#       IdentitySet([<User('kolia','Cool      Kolian[S.A.]',      'kolia$$$')>,
<User('zina','Zina Korzina', 'zk18')>])
```

Метод commit() фиксирует транзакцию, которая до того была в процессе, отправляя все оставшиеся изменения в базу:

```
session.commit()
```

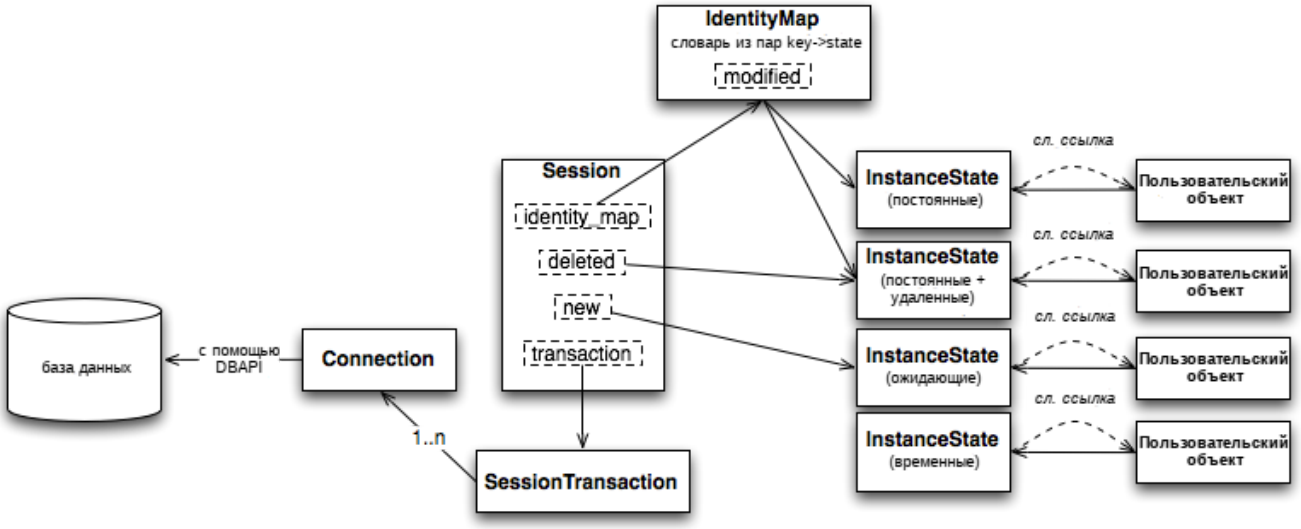
Ресурсы подключений, что использовались в сессии, снова освобождаются и возвращаются в набор. Последовательные операции с сессией произойдут в новой транзакции, которая снова запросит себе ресурсы по первому требованию. Атрибуту id, который раньше был None, присвоено значение:

```
print('User ID:', admin_user.id)     # User ID: 1
```

После того, как сессия добавит новые записи в базу, все только что созданные идентификаторы будут доступны в объекте, немедленно или по первому требованию. В данном случае при обращении к объекту была перезагружена целая запись, так как после вызова commit() началась новая транзакция. SQLAlchemy обновляет данные от предыдущей транзакции при первом обращении с новой транзакцией, так что пользователю доступно самое последнее ее состояние.

Сессия

На рисунке приведена схема взаимодействия сессии (Session) с базовыми структурами SQLAlchemy.



Общедоступными объектами на рисунке выше является сам объект Session, а также коллекция пользовательских объектов, каждый из которых является экземпляром класса, используемого для создания отображения. Здесь мы можем увидеть, что используемые для отображения объекты ссылаются на конструкцию из состава SQLAlchemy с именем InstanceState, которая отслеживает состояние отдельного объектно-реляционного отображения, включая ожидающие операции изменения атрибутов, а также факт истечения срока действия атрибутов. Объект InstanceState является инструментарием для работы с атрибутами на уровне экземпляра класса, описанным в предыдущем разделе под названием “Анатомия отображения” и соответствующим объекту ClassManager на уровне класса, который позволяет поддерживать состояние словаря используемого для создания отображения объекта (т.е., атрибута __dict__, описанного в рамках языка программирования Python) на стороне ассоциированных с классом объектов AttributeImpl.

Отслеживание состояния

Объект IdentityMap позволяет создавать отображение индивидуальных данных базы данных для объектов InstanceState, которые в свою очередь используются теми объектами, которым требуются эти индивидуальные данные, называемые также постоянными (persistent). Стандартная реализация объекта IdentityMap взаимодействует с объектом InstanceState для самостоятельного управления объемом занятой памяти путем удаления созданных пользователем отображений в тех случаях, когда удаляются все жесткие ссылки на эти отображения - таким образом, этот объект функционирует аналогично объекту WeakValueDictionary из состава Python. Объект Session защищает набор всех объектов с пометкой “устаревший” (“dirty”) или “удаленный” (“deleted”), а также охраняет объекты с пометкой “новый” (“new”) от механизма сборки мусора путем создания жестких ссылок на эти объекты в случае ожидания их изменений. Все жесткие ссылки удаляются после выполнения операции сохранения данных.

Объект InstanceState также выполняет критичную задачу, заключающуюся в поддержании “списка изменений” для атрибутов определенного объекта с использованием системы перемещения данных при изменении, которая сохраняет “данные предыдущего состояния” определенного атрибута в словаре с именем commited_state перед использованием переданного значения для изменения значения в словаре атрибутов объекта. Во время выполнения операции сохранения изменений содержимое словаря commited_state, а также ассоциированного с объектом словаря __dict__ сравниваются с целью формирования набора измененных данных для каждого из объектов.

Контроль транзакций

Объект Session при обычном сценарии использования поддерживает открытую транзакцию для выполнения всех операций, которая завершается в момент вызова метода commit или rollback. Объект SessionTransaction поддерживает набор объектов Connection, который может быть как пустым, так и заполненным, причем каждый объект в нем представляет открытую транзакцию для определенной базы данных. Объект SessionTransaction является объектом с отложенной инициализацией, которая начинается при отсутствии данных состояния базы данных. Так как определенная база данных должна участвовать в процессе выполнения запроса, соответствующий этой базе данных объект соединения Connection добавляется в список соединений объекта SessionTransaction. Хотя обычно в каждый момент времени используется одно соединение с базой данных, поддерживается сценарий использования множества соединений, в котором определенное соединение используется для выполнения определенной операции, в соответствии с ассоциированными с объектами Table, Mapper данными конфигурации, либо в соответствии с конструкциями языка SQL, применяемыми в рамках операции. При использовании множества соединений также может координироваться процесс выполнения транзакции при применении двухфазной схемы в тех случаях, когда реализация DB-API предоставляет ее.

Состояния сессии

Понимание состояний сессии может быть полезно для предотвращения исключений и обработки неопределенного поведения. Существует 4 состояния для объектов данных:

- временное (Transient) - объект вне сессии и вне базы данных;
- в ожидании (Pending) - объект был добавлен в сессию через add(), но не был сохранён в БД;
- постоянное (Persistent) - объект имеет соответствующую запись в БД;
- отключён (Detached) - объект отключён от сессии, но имеет запись в БД.

Резюме

- Вызов sessionmaker() необходимо выполнить только один раз, желательно в глобальном пространстве имён.
- Сессию необходимо отделять от функций и объектов (передавать её как параметр).
- Важно понимать, где начинается и где заканчивается транзакция; делать транзакции короткими (завершать их после серии операций, а не держать открытыми).

Базы данных и тестирование

Укажем некоторые особенности тестирования кода, работающего с базами данных:

1. Для тестирования простых ситуаций (создание схемы БД, добавление данных, удаление данных/таблиц) имеет смысл использовать БД в памяти (для ускорения тестов):

```
@pytest.yield_fixture
def db():
    with sqlite.connect(':memory:') as db:
        yield db
```

2. Имеет смысл создавать фикстуры для работы с конкретной СУБД:

```
@pytest.yield_fixture
def redis():
    with Redis() as redis:
        yield redis

@pytest.yield_fixture
def db():
    with sqlite.connect(':memory:') as db:
        yield db

def test_a(db, redis):
    db.execute(...)
    redis.set(...)
```

3. Имеет смысл сделать заготовку базы с тестовыми данными (можно обратить внимание на пакет python-testdata).
4. Имеет смысл иметь заготовку данных для разных тестовых сценариев.
5. Каждый тест должен быть изолирован от других и работать на оригинальных данных (применять откат транзакций между тестами):

```
@pytest.yield_fixture()
def db_transaction(request):
    orm.session.begin()
    yield orm.session
    orm.session.rollback()
```

Итоги

С использованием SQLAlchemy работать с базой данных становится так же удобно как и со структурами языка программирования. Но при этом всё равно SQLAlchemy подразумевает, что **разработчик должен знать и понимать, как работает SQL**. В современных фреймворках для разработки сайтов так или иначе используется ОРМ, в Django - это своя реализация django-orm, но во множестве других, таких как flask, pyramids в основном используется как раз SQLAlchemy.

Помимо SQLAlchemy можно также обратить внимание на другие ORM: PeeWee, Pony ORM.

Домашнее задание

1. Реализовать класс **Хранилище** для клиента и сервера. Хранение необходимо осуществлять в базе данных. В качестве базы данных можно выбрать любую СУБД (sqlite, PostgreSQL, MySQL и прочие). Для взаимодействия с БД можно использовать ORM.

В качестве опорной схемы базы данных предлагается следующий вариант.

- На стороне сервера БД содержит следующие таблицы:
  - клиент:
    - логин;
    - информация.
  - история_клиента:
    - время входа;
    - ip-адрес.
  - список_контактов (составляется на основании выборки всех записей с id_владельца)
    - id_владельца;
    - id_клиента.
- Реализовать хранение информации в БД на стороне клиента:
    - список_контактов;
    - история_сообщений.
- Реализовать функционал работы со списком контактов по протоколу JIM:

Получение списка контактов

Запрос к серверу:

```
  {
    "action": "get_contacts",
    "time": <unix timestamp>,
   }
```

Положительный ответ сервера будет состоять из нескольких частей. Первая часть содержит код результата и количество контактов текущего пользователя:

```
{
    "response": 202,
    "quantity": xxx          # количество контактов
}
```

Далее сервер отсылает xxx сообщений формата:

```
{
    "action": "contact_list",
    "user_id": "nickname"
}
```

Получение списка контактов - не самая частая операция при взаимодействии с сервером. Она должна выполняться после подключения (и авторизации) клиента. Инициируется клиентом. В процессе получения списка контактов клиенту не допускается инициировать другие запросы.

Добавление/удаление контакта в список контактов:

Запрос к серверу:

```
{
    "action": "add_contact" | "del_contact",
    "user_id": "nickname",
    "time": <unix timestamp>,
}
```

Ответ сервера будет содержать одно сообщение с кодом результата и не обязательной расшифровкой:

```
{
    "response": xxx,
}
```

- Для работы со списком контактов предлагается реализовать дополнительные классы:
    - **СписокКонтактов** - класс, реализующий операции с контактами (добавление, удаление);
    - **КонтактКонтроллер** - класс, реализующий взаимодействие классов СписокКонтактов и СписокКонтактовGUI;
    - **СписокКонтактовGUI** - базовый класс для отображения списка контактов (консольный, графический, WEB)

2. * Реализовать возможность создавать чат для нескольких пользователей (группа):
    - хранение информации о группах в БД сервера;
    - отправка сообщений пользователям группы.

Дополнительные материалы

1. Slideshare. Michael Bayer. Introduction to SQLAlchemy
2. Slideshare. Relational Database Access with Python
3. Slideshare. Introduction to SQLAlchemy by Jorge A. Medina
4. ORM. Использование SQLAlchemy
5. Вводная по сложным запросам в SQLAlchemy
6. Python. Работа с базой данных, часть 1/2. Используем DB-API
7. PEP 249 – Python Database API Specification v2.0
8. The Novice’s Guide to the Python 3 DB-API
9. SQLAlchemy - как втянуться

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Myers Jason, Copeland Rick. Essential SQLAlchemy: Mapping Python to Databases
2. Michael Driscoll. Python 101 (Chapter 34 - SQLAlchemy)
3. Лутц Марк. Программирование на Python, том II, 4-е издание.
4. Бизли Дэвид. Python. Подробный справочник.
5. Wiki Портала Python-программистов. SQLAlchemy
6. SQLAlchemy Architecture. Michael Bayer
7. Глава 20 из книги “Архитектура приложений с открытым исходным кодом”, том 2