**Лабораторная работа №7**

**Организация взаимодействия с базой данных через приложение, использующее технологию ORM**

**Цель работы:** разработать приложение, использующее технологию ORM, для взаимодействия с базой данных.

**Основные теоретические сведения**

*ORM* (англ. Object-Relational Mapping) – это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Существуют как проприетарные, так и свободные реализации этой технологии.

Благодаря этой технологии разработчики могут использовать язык программирования, с которым им удобно работать с базой данных, вместо написания операторов SQL или хранимых процедур. Это может значительно ускорить разработку приложений, особенно на начальном этапе. *ORM* также позволяет переключать приложение между различными реляционными базами данных. Например, приложение может быть переключено с MySQL на PostgreSQL с минимальными изменениями кода.

Так, например, *Peewee* является легковесной реализацией технологии *ORM* на языке python, которая в данный момент поддерживает такие СУБД, как ***postgresql*, *mysql*** и *sqlite*. Разумеется, это не единственная *ORM*-библиотека для разработчиков *Python*. Фреймворк *Django* предоставляет собственную *ORM*-библиотеку, кроме этого, существует ***SqlAlchemy***. Любая библиотека *ORM* использует понятие моделей в проектировании. Модель – это класс *Python*, соответствующий таблице в базе данных, а его свойства – это столбцы таблиц.

*Flask* – фреймворк для создания веб-приложений на языке программирования *Python*, использующий набор инструментов «*Werkzeug*», а также шаблонизатор *Jinja2*. Относится к категории так называемых микрофреймворков– минималистичных каркасов веб-приложений, предоставляющих лишь самые базовые возможности.

Сериализация (в программировании) — процесс перевода структуры данных в последовательность байтов. Обратной к операции сериализации является операция десериализации (структуризации) — создание структуры данных из битовой последовательности.

Сериализация используется для передачи объектов по сети и для сохранения их в файлы. Например, нужно создать распределённое приложение, разные части которого должны обмениваться данными со сложной структурой. В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет сериализацию и десериализацию.

*REST* (от англ. Representational State Transfer – «передача состояния представления») – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Другими словами, *REST* – это набор правил о том, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

***REST API*** предполагает предоставление доступа к своим данным серверным приложением клиентскому приложению по определенному URL. *REST API* позволяет использовать для общения между программами **протокол HTTP** (версия с использованием шифрования – HTTPS), с помощью которого мы получаем и отправляем большую часть информации в интернете.

Маршрут (англ. Route) – это «имя», которое отсылает работу API к определенным эндпоинтам. Проще говоря, маршрут – это URL, к которому можно обратиться разными HTTP методами. Маршрут может иметь несколько эндпоинтов.

Эндпоинт (англ. Endpoint) – это само обращение к маршруту отдельным HTTP методом. Эндпоинты выполняют конкретную задачу, принимают параметры и возвращают данные Клиенту.

**Задание к работе**

1. Изучить технологию ORM.

2. Разработать снабженное графическим интерфейсом приложение, которое обеспечит подключение к базе данных, разработанной на основе предыдущих лабораторных работ, а также обеспечит выполнение запросов с использованием технологии ORM.

**Пример выполнения задания**

Для разработки приложения рекомендуется разбивать отдельные его функциональные части на файлы, которые будут иметь разное назначение. Создадим файлы config.py, models.py, serializers.py, services.py, routers.py, app.py.

В файле config.py будут описаны объект приложения Flask и объект, предоставляющий инструменты для взаимодействия с базой данных.

В файле models.py с использованием объявленной в config.py базы данных будут объявлены базовые модели и все модели, необходимые для работы приложения.

В файле serializers.py будут объявлены сериалайзеры для конвертации моделей в словари.

В файле services.py будут реализованы функции, выполняющие действия над моделями (создание и получение списков записей и отдельных записей).

В файле routers.py будут объявлены все необходимые эндпоинты, через которые будет реализовано взаимодействие с сервисами.

В файле app.py будет импортировано само приложение, уже содержащее все необходимые маршруты, а также будет осуществляться запуск этого приложения.

Файл config.py. В коде ниже импортируется ORM peewee и микрофреймворк Flask, предназначенный для разработки REST API.

**from** peewee **import** \*

**from** flask **import** Flask

Создание Flask-приложения и загрузка в него стандартных настроек.

app = Flask(\_\_name\_\_)

*# Загрузка стандартных настроек*  
app.config.from\_object(\_\_name\_\_)

Подключение к базе данных и создание класса базовой модели (базового класса), от которого будут наследоваться все остальные модели.

*# Инициализация БД*db = SqliteDatabase(**'example.db'**)  
  
**class** BaseModel(Model):  
 **class** Meta:  
 database = db

Файл models.py. В коде ниже представлено объявление всех используемых в приложении моделей. Создание для них таблиц.

**import** peewee **as** pw  
**from** config **import** BaseModel, db  
  
**class** Contact(BaseModel):  
 id = pw.IntegerField(primary\_key=True)  
 fcs = pw.CharField(max\_length=255)  
 phone = pw.CharField(max\_length=25)  
 address = pw.CharField(max\_length=255)  
  
  
**class** Customer(BaseModel):  
 id = pw.IntegerField(primary\_key=True)  
 contact = pw.ForeignKeyField(Contact, on\_delete=**'CASCADE'**, null=False, backref=**'customers'**)

……. (здесь описываем все таблицы)

*# Создание таблиц для всех моделей*

**with** db:  
 db.create\_tables([  
 Contact, Qualification, Executor, Customer, ContractStatus, JobType, Contract  
 ])

Файл serializers.py. В коде ниже представлен пример объявления сериалайзера для таблицы «customer».

**def** serialize\_customer(model):  
 **return** {  
 **'id'**: model.id,  
 **'contact'**: {  
 **'id'**: model.contact.id,  
 **'fcs'**: model.contact.fcs,  
 **'phone'**: model.contact.phone,  
 **'address'**: model.contact.address,  
 }  
 }

Файл services.py. Сервисы (в нашем случае) представляют собой функции, обеспечивающие удобный интерфейс для взаимодействия с базой данных. Импортируем модели, используемые в приложении, а также функцию jsonify(), переводящую объекты языка Python (например, словари и списки) в формат json.

**from** . **import** models  
**from** flask **import** jsonify  
**from** .serializers **import** \*

Функция where\_filters() принимает запрос, модель, а также фильтры, полученные из эндпоинта. Она преобразует эти данные в формат, используемый ORM peewee, и возвращает запрос, содержащий where.

**def** where\_filters(query, model: models.BaseModel, \*\*filters):  
 \_filters = [  
 getattr(model, key) == value  
 **for** key, value **in** filters.items() **if** value **is not** None  
 ]  
 **if** \_filters:  
 **return** query.where(\*\_filters)  
 **return** query

Ниже представлены вспомогательные функции для получения списка записей и одной конкретной записи (строки таблицы). На вход функция execute\_get\_all() получает модель, над которой нужно произвести действия, сериалайзер для неё и фильтры. Если необходимо получить конкретную запись, то используется функция execute\_get\_one(), в которую передается первичный ключ, модель и сериалайзер.

**def** execute\_get\_all(model, serializer, \*\*filters):  
 query = model.select()  
 query = where\_filters(query, model, \*\*filters)  
 **return** jsonify([serializer(model) **for** model **in** query])  
  
**def** execute\_get\_one(pk, model, serializer):  
 **return** serializer(model.select().where(model.id == int(pk)).get())

Это функции для получения конкретной записи, списка записи и создания новой записи на примере «customer».

**def** get\_customer\_detail(pk: int):  
 **return** execute\_get\_one(pk, models.Customer, serialize\_customer)  
  
**def** get\_customers(\*\*filters):  
 **return** execute\_get\_all(models.Customer, serialize\_customer, \*\*filters)  
  
**def** create\_customers(json: dict):  
 **return** serialize\_customer(models.Customer.create(\*\*json))

Файл routers.py. В коде ниже производится импорт приложения, для которого будут создаваться эндпоинты, объекта request для получения json-файлов в POST-запросах, а также используемых сервисов.

**from** config **import** app  
**from** flask **import** request  
**from** . **import** services

Ниже представлены эндпоинты для создания новой записи на примере модели «customer», получения списка записей и конкретной записи.

@app.route(**'/customers'**, methods=[**'POST'**])  
**def** post\_customers():  
 **if** request.is\_json:  
 **return** services.create\_customers(request.get\_json())  
 **return** {}  
  
@app.route(**'/customers'**, methods=[**'GET'**])  
**def** get\_customers():  
 **return** services.get\_customers()  
  
@app.route(**'/customers/<pk>'**, methods=[**'GET'**])  
**def** get\_customer\_detail(pk: int):  
 **return** services.get\_customer\_detail(pk)

Главный файл app.py. В нем производится запуск приложения.

**from** src.routers **import** app  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run()

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

Рис. 6. Снимки экрана выполнения программы

В процессе выполнения лабораторной работы были получены навыки разработки приложения для взаимодействия с базой данных, использующего технологию ORM.