Кафедра интеллектуальных информационных технологий

Отчет по лабораторной работе №3
по курсу
«Проектирование баз знаний»

на тему: «Разработка и выравнивание онтологий»

Выполнили студенты группы 921702: Даниленко В.С. Драгун В.С.

Проверил: Ковалёв М.В.

МИНСК 2022

Цель работы:
Приобрести навыки разработки онтологий предметных областей
Задачи:

Необходимо разработать графическое или web-приложение, использующее одно из хранилищ онтологий по выбранной предметной области.

Используемые технологии:

- как rdf хранилище был выбран AllegroGraph
- Python
- FastApi
- SPARQL
- docker

Причины почему были выбраны данные технологии:

Python

На данный момент Python является самым популярным языком программирования. Причинами всего этого являются его простота и большое количество мощных, постоянно обновляющихся инструментов. Преимущества Python перед другими языками программирования(C#, C++, Java)

- Динамическая типизация
- Простота
- Интерпретируемость
- Широкий спектр технологий
- Единый стандарт написание кода PEP

AllegroGraph

AllegroGraph – современная, высокопроизводительная, устойчивая графическая база данных с закрытым исходным кодом. AllegroGraph совмещает эффективное использование оперативной памяти с использованием дисковых хранилищ, что позволяет ему масштабировать миллиарды четверок (quads), сохраняя при этом высокую производительность. Поддерживает языки запросов к данным SPARQL, RDFS + + и Prolog и автоматически использует те из них, которые совместимы с приложениями пользователя. AllegroGraph разрабатывает фирма Franz Inc, которая широко известна благодаря таким продуктам, как TopBraidComposer (редактор онтологий), RacerPro (OWL DL онтология) и др.

Все особенности:

- Полное соответствие требованиям ACID и поддержка операций фиксации, отката, и контрольных точек(Commit, Rollback, and Checkpointing.);
- Полная и быстрая восстанавливаемость;
- Полная одновременность чтения, почти полная одновременность записи;
- Создание резервных копий онлайн, задание точки восстановления системы (Point-in-Time Recovery), репликация, теплый резерв (Warm Standby);
- Динамическая и автоматическая индексация - все зафиксированные тройки всегда индексируются;
- Расширенное индексирование текста - индексирование текста на предикат;
- Объединение в одно целое платформы SOLR и СУБД MongoDB;
- Поддерживает языки Java Sesame, Java Jena, Python, C#, Clojure, Perl, Ruby, Scala и Lisp;
- Симметричная мультипроцессорность (SMP) - автоматическое управление ресурсами (Automatic Resource Management) для всех процессоров и дисков и оптимизированное использование памяти;
- Сжатие индексов на основе столбцов - снижение подкачки, повышение производительности;
- Тройной уровень безопасности с фильтрами безопасности;
- Облачный AllegroGraph - Amazon EC2;
- RDF - сервер AllegroGraph может быть создан с помощью языка JavaScript;
- Интерфейс на основе JavaScript (JIG) для общего обхода графов;
- Поддержка алгоритма Soundex (сравнение двух строк по их звучанию) позволяет свободно выполнять индексирование текста на основе фонетического произношения;
- Показатели, задающиеся пользователем, полностью контролируемые системным администратором;
- Клиент-серверная архитектура GRUFF с графическим построителем запросов;
- Подключаемый интерфейс для текстовых индексаторов;
- Специализированные (dedicated) и открытые сеансы - в специализированных (dedicated) сеансах пользователи могут работать со своими собственными наборами правил для одной и той же базы данных.

FastApi

FastAPI — это фреймворк для создания лаконичных и довольно быстрых HTTP API-серверов со встроенными валидацией, сериализацией и асинхронностью. Стоит он на плечах двух других фреймворков. Работой с web в FastAPI занимается Starlette, за валидацию отвечает Pydantic.

Комбайн получился лёгким, не перегруженным и более чем достаточным по функционалу.

Starlette — новый, шустрый и классные фреймворк, реализующий подход ASGI. В нем всё заточено на асинхронность и новые фишки 3-й ветки Python. Кроме этого в Starlette есть ещё целая пачка серьёзных плюшек:

- GraphQL из коробки
- Веб-сокеты уже встроены и готовы к работе
- Готовый набор миддлверов для работы с авторизацией/аутентификацией, CORS
- Встроенные асинхронные таски

Асинхронное программирование — это потоковая обработка программного обеспечения /пользовательского пространства, где приложение, а не процессор, управляет потоками и переключением контекста. В асинхронном программировании контекст переключается только в заданных точках переключения, а не с периодичностью, определённой CPU.

Когда вы запускаете что-то асинхронно, это означает, что оно не блокируется, вы выполняете его, не дожидаясь завершения, и продолжаете выполнять другие вещи. Параллелизм означает выполнение нескольких задач одновременно, параллельно. Параллелизм хорошо работает, когда вы можете разделить задачи на независимые части работы.

Основные возможности FastAPI

FastAPI — это, по сути, нашлёпка на родные классы Starlette, добавляющая пачку новых фич к уже и так неплохому фреймворку.

- Плюшки для создания REST API сервисов + Swagger-документация для методов. Starlette ориентируется на модный GraphQL, FastAPI заботится о тех, кто пилит REST.
- Удобные примочки, построенные на подсказках типов переменных. Например, встроенные валидаторы данных.
- Приятные полезности для процессов авторизации и аутентификации — поддержка JWT, OAuth2.

Важные причины выбрать FastAPI:

1. Асинхронность
2. Типизированность
3. Встроенная документация (Swagger)
4. Применение websockets

SPARQL

SPARQL (рекурсивный акроним от англ. *SPARQL Protocol and RDF Query Language*) — язык запросов к данным, представленным по модели RDF, а также протокол для передачи этих запросов и ответов на них. SPARQL является рекомендацией консорциума W3C и одной из технологий семантической паутины. Предоставление SPARQL-точек доступа (англ. *SPARQL-endpoint*) является рекомендованной практикой при публикации данных во всемирной паутине.

SPARQL позволяет пользователям писать глобально однозначные запросы. Например, следующий запрос возвращает имена и адреса каждого человека в мире:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
 ?person a foaf:Person.
 ?person foaf:name ?name.
 ?person foaf:mbox ?email.
}
```

Приведённые параметры используются для описания человека, включенного в FOAF. Это иллюстрирует видение Семантической паутины как единой огромной базы данных. Каждый идентификатор в SPARQL, URI, глобально однозначен, в отличие от «email» или «e-mail», обычно используемых в SQL.

Этот запрос может быть распределен на несколько конечных точек SPARQL, разных компьютеров, и сбор результатов осуществляется процедурой, известной как федеративный поиск

Формы запросов

Язык SPARQL определяет четыре различных варианта запросов для различных целей:

SELECT запрос

Извлекает необработанные значения из точки доступа SPARQL и возвращает результаты в формате таблицы.

CONSTRUCT запрос

Извлекает информацию из точки доступа SPARQL в формате RDF и преобразовывает результаты к определенной форме.

ASK запрос

Формирует запрос типа Истина/Ложь.

DESCRIBE запрос

Получает описание RDF-ресурса. Реализация поведения DESCRIBE-запросов определяется разработчиком точки доступа SPARQL.

Каждая из этих форм запроса содержит блок WHERE для указания ограничений, хотя в случае запроса DESCRIBE этот блок не является обязательным.

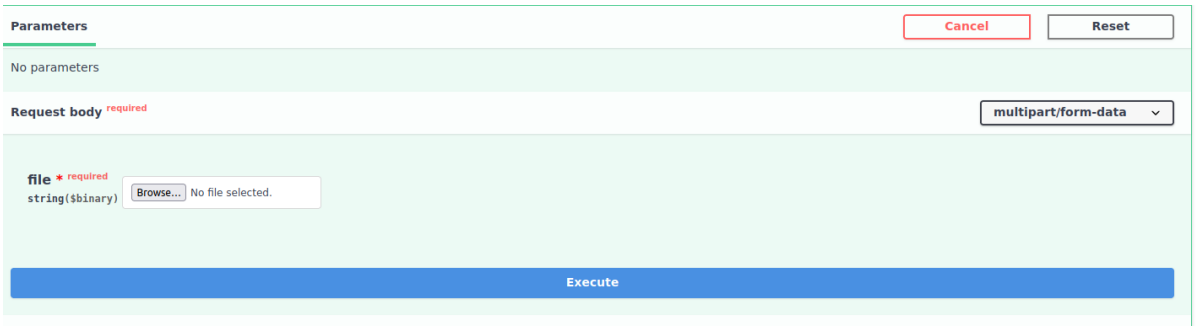
Описание разработанного приложения

Приложение разработано при помощи языка программирования Python. Данное веб приложения основано на архитектуре клиент-сервер. В роли клиента выступает браузер, а в роли приложение разработанное при помощи FastApi. Общение между клиентом и сервером происходит при помощи REST запросов(GET, POST, DELETE и т.д).

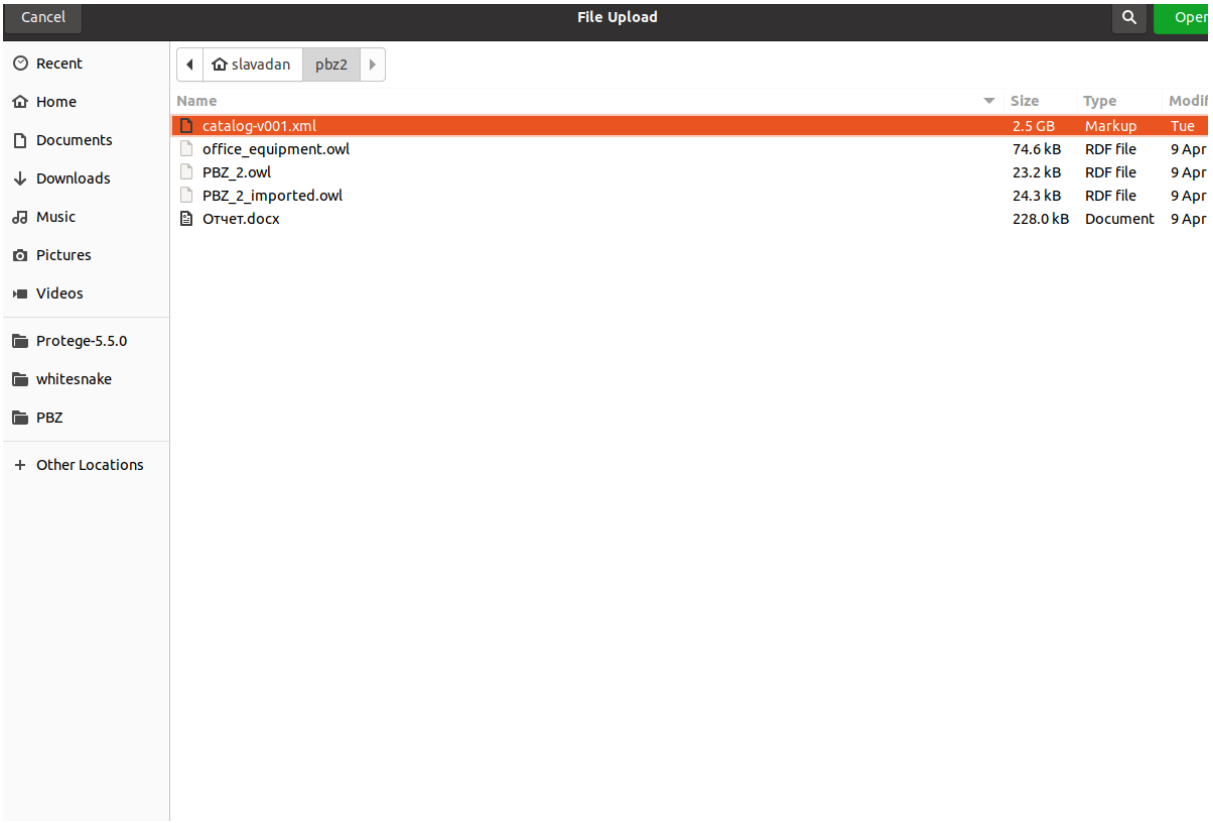
Как основная база данных используется AllegroGraph, который также представляет api для работы с онтологиями.

Данное приложение позволяет работать с любыми онтологиями, которые можно загружать через owl файл.

Пример добавления своей онтологии в базу данных:



нажимаем кнопку browse



выбираем файл формата owl(в данном случае был выбран файл с названием PBZ_2.owl).
После этого происходит загрузка нашего файла в бд.

До загрузки файла:

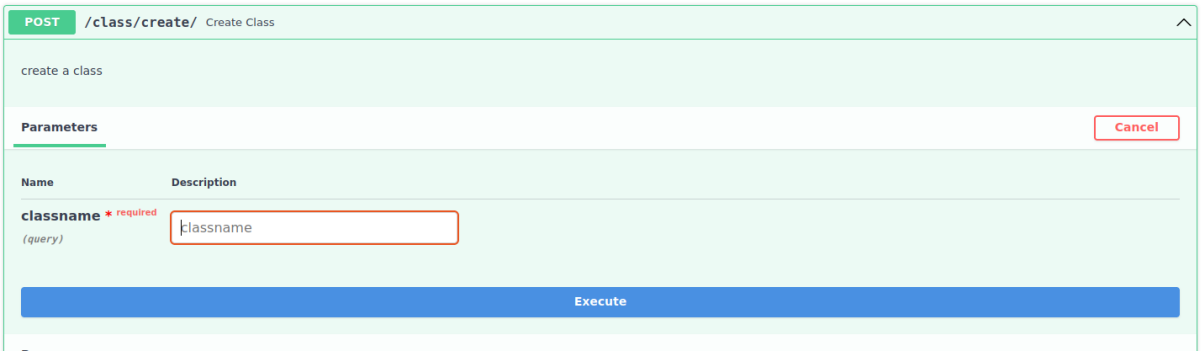
После загрузки файла:

s	p	o
pbz_2	rdf:type	owl:Ontology
has_form_of	rdf:type	owl:ObjectProperty
date_of_issue	rdf:type	owl:DatatypeProperty
Air_Stamp	rdf:type	owl:Class
Air_Stamp	rdfs:subClassOf	By_means_of_transportation
Automotive_Mark	rdf:type	owl:Class
Automotive_Mark	rdfs:subClassOf	By_means_of_transportation
By_means_of_transportation	rdf:type	owl:Class
By_means_of_transportation	rdfs:subClassOf	Postage_stamp
Commemorative_stamp	rdf:type	owl:Class
Commemorative_stamp	rdfs:subClassOf	General_purpose_mark
Custom_brand	rdf:type	owl:Class
Custom_brand	rdfs:subClassOf	General_purpose_mark
Express_mail_stamp	rdf:type	owl:Class
Express_mail_stamp	rdfs:subClassOf	General_purpose_mark
Ferry_stamp	rdf:type	owl:Class

После этого мы можем делать разные операции с данной онтологией.
Главное меню со списком операций:

Пример операции создания класс в загруженной онтологии:

1) Выбираем нужную операцию:



2) Вводим название нашего нового класса и нажимаем кнопку выполнить

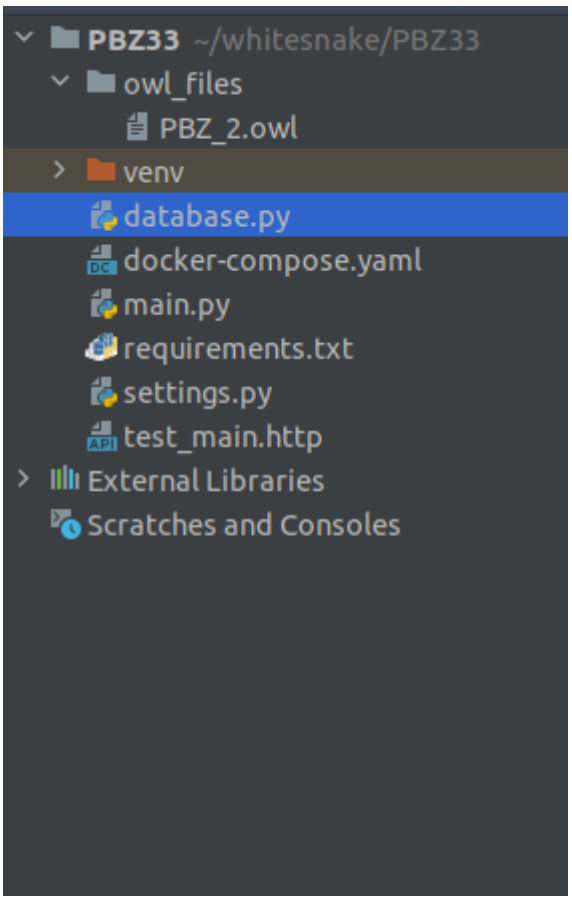
После этого мы можем увидеть статус код 201, что означает CREATED(создано), в случае неудачной операции вернется статус код 400е или 500е, в зависимости где эта ошибка произошла(400 - клиент, 500 - сервер)

Результат наших удачных операций можно посмотреть 2мя способами.

1) Выполнить операцию GET из главного меню

2) Зайти в интерфейс, который предоставляет нам AllegroGraph и ввести SPARQL запрос на получение конкретного или всех элементов.

Перейдем к программной части нашего приложения.
Структура проекта:



- database.py - в данном файле реализованы, методы для работы с базой данных
- settings.py - в данном файле находятся параметры, которые используются для подключения к бд и тд.
- main.py - запускает наше приложени, и так же содержит все эндпоинты, по которым передаются запросы
- requirements.txt - файл, который можно посмотреть какие зависимости использует наше приложение
- docker-compose.yaml - файло для управления, контейнерами. В нашем случае содержиться только 1 контейнер с базой данных

settings.py

```
REPOSITORY_NAME = "repo"

CATALOG_NAME = ''

HOST = 'localhost'

PORT = '10035'

USER = 'test'

PASSWORD = 'xyzzy'

OWL_FILES_STORAGE = './owl_files/'
```

docker-compose.yaml

```
version: '3.8'

services:
  db:
    image: franzinc/agraph
    expose:
      - 5432
    environment:
      - AGRAPH_SUPER_USER=test
      - AGRAPH_SUPER_PASSWORD=test
```

main.py

```python
import uvicorn
from fastapi import FastAPI, UploadFile, File
from fastapi.responses import FileResponse, JSONResponse
from starlette import status
from typing import Optional
import database


app = FastAPI()


@app.post("/file/upload/")
async def root(file: Optional[UploadFile] = File(...)):...


@app.get("/class/")
async def get_classes():...


@app.get("/subclasses/")
async def get_subclasses():...


@app.get("/object_property/")
async def get_object_property():...


@app.get("/data_property/")
async def get_data_properties():...


@app.post("/data_property/create/")
async def create_data_property(data_property):...


@app.post("/data_property/connect/")
async def add_data_property_to_class(subject, data_property, object_class):...
```

```python
@app.delete("/object_property/delete/")
async def delete_object_property(object_property):
    """delete object property"""
    if database.execute_delete_query(f":{object_property}", "rdf:type", "owl:ObjectPropert
        return JSONResponse(
            status_code=status.HTTP_201_CREATED,
            content={}
        )

    return JSONResponse(status_code=status.HTTP_400_BAD_REQUEST, content={})


def runserver(host="localhost", port=8888):
    uvicorn.run(
        "main:app",
        host=host,
        port=port,
        lifespan="on",
    )


if __name__ == "__main__":
    runserver()
```

database.py

```python
server = AllegroGraphServer(
    host=settings.HOST, port=settings.PORT,
    user=settings.USER, password=settings.PASSWORD
)

catalog = server.openCatalog(settings.CATALOG_NAME)

repository = catalog.getRepository(settings.REPOSITORY_NAME, Repository.ACCESS)


def add_file_to_rep(filename: str):
    with repository.getConnection() as connection:
        connection.addFile(settings.OWL_FILES_STORAGE + filename)


async def write_file(file: Optional[UploadFile]) -> bool:
    if file is None:
        return False

    with open(settings.OWL_FILES_STORAGE + file.filename, "wb") as created_file:
        content = await file.read()
        created_file.write(content)
        created_file.close()
        add_file_to_rep(file.filename)

    return True
```

самый важный файл, который предоставляет функционал для работы с бд.
server - переменная, которая является основным нашим соединением с бд
repository - репозиторий с которым мы работаем.
ВАЖНО - параметры беруться из файла settings.py

методы add_file_to_rep и write_file отвечают за загрузку файла в бд

```python
def execute_get_query(subject="?s", relation="?r", object="?o"):
    """select query for get endpoints"""
    query_string = "SELECT ?s ?r ?o WHERE {%s %s %s}" % (subject, relation, object)
    result_list = []

    with repository.getConnection() as connection:
        result = connection.executeTupleQuery(
            query=query_string
        )

        with result:
            for bindung_set in result:
                result_list.append(
                    {
                        "subject": bindung_set.getValue('s').__str__(),
                        "relation": bindung_set.getValue('r').__str__(),
                        "object": bindung_set.getValue('o').__str__()
                    }
                )

    return result_list


def execute_post_query(subject, relation, object):
    string_query = "INSERT DATA { %s %s %s}" % (subject, relation, object)

    with repository.getConnection() as connection:
        return connection.executeUpdate(
            query=string_query
        )


def execute_delete_query(subject, predicate, object):
    string_query = "DELETE DATA { %s %s %s }" % (subject, predicate, object)

    with repository.getConnection() as connection:
        return connection.executeUpdate(
            query=string_query
        )
```

3 метода которые служат для вставки, удаления, поиска в бд.
Данные операции основаны на запросе SPARQL который находится в переменной query_string, которая также подхватывает параметры которые получает наши методы при их вызове.

Источники:

- https://franz.com/agraph/support/documentation/6.4.0/python/api.html - документаци по api для работы с бд
- https://hub.docker.com/r/franzinc/agraph/ - предоставляет образы для контейнера с базой данных
- https://fastapi.tiangolo.com/ru/ - документация fastapi на русском языке
- https://pypi.org/ - предоставляет инструменты для python(например api для работы с бд)
- https://dic.academic.ru/dic.nsf/ruwiki/102698#.D0.9F.D1.80.D0.B5.D0.B8.D0.BC.D1.83.D1.89.D0.B5.D1.81.D1.82.D0.B2.D0.B0 - информация о SPARQL