|  |  |  |
| --- | --- | --- |
|  | **Софийски университет „Св. Кл. Охридски”**  Факултет по математика и информатика  *Бакалавърска програма „Анализ на данни”* |  |

**Дисциплина: Езици и технологии на метаданни**

***Летен семестър, 2024/2025 год.***

**Тема №8: „SPARQL Query Builder“**

**Курсов проект**

*Автори:*

*Дарин Крумов, фак. номер 0MI0200003*

*Самуил Ганев, фак. номер 6MI0200001*

юни, 2025 г.

София

Съдържание

[1 Въведение 3](#intro)

[2 Анализ на решението 3](#analiz)

[2.1 Работен процес 3](#analiz_1)

[2.2 Тип и представяне на съдържанието 5](#analiz_2)

[2.3 Използване на изкуствен интелект (ИИ) за разработването на проекта 7](#analiz_3)

[3 Дизайн 8](#design)

[4 Тестване 13](#tests)

[5 Заключение и възможно бъдещо развитие 16](#future_work)

[6 Разпределение на работата 16](#work_distribution)

[7 Използвани литературни източници и Уеб сайтове 17](#lit)

[8 Апендикс 18](#apendix)

# Въведение

Настоящият документ описва курсовия проект „SPARQL Query Builder“, разработен в рамките на дисциплината „Езици и технологии на метаданни“. Актуалността на темата произтича от нарастващата важност на Семантичния уеб и свързаните данни, както и от сложността на езика за заявки SPARQL, който често представлява бариера за потребители без задълбочени технически познания.

Проектът решава проблема с трудното конструиране, изпълнение и разбиране на SPARQL заявки, особено към големи и комплексни хранилища на данни като Wikidata. Контекстът на проблема е необходимостта от по-достъпни инструменти, които позволяват на изследователи, анализатори на данни и други заинтересовани страни да извличат информация от SPARQL крайни точки (endpoints) без да са експерти в самия език SPARQL или в спецификите на различните клиенти за заявки.

Този проблем се решава чрез разработката на уеб-базирано приложение, използващо Python библиотеката Streamlit, която пък позвоява бързото изграждане на интерактивен потребителски интерфейс, който предоставя поле за въвеждане и редакция на SPARQL заявки, опции за избор на ендпойнтове, вградени шаблони (специално за Wikidata), функционалност за търсене на Wikidata ентитита (класове и предикати), както и визуализация на резултатите в табличен вид. По този начин технологията Streamlit улеснява създаването на интуитивен инструмент, който скрива част от сложността на SPARQL взаимодействието.

Останалата част от документа е структурирана както следва: Секция 2 представя анализ на решението, включващ работния процес, типа на използваното съдържание и използването на изкуствен интелект от нас. Секция 3 описва дизайна на приложението, архитектурата и използваните технологии. Секция 4 детайлизира процеса на тестване и финализиране. Секция 5 обобщава резултатите и описваме бъдещо развитие. Секция 6 описва разпределението на работата между нас като автори. Секция 7 изброява използваните източници.

# Анализ на решението

## Работен процес

Работният процес на приложението SPARQL Query Builder може да бъде описан в следните стъпки: вход, обработка и изход.

* **Входно съдържание/данни и откъде се получава:**
  + **SPARQL Заявка:** Текстово съдържание, въведено или модифицирано от потребителя в специално текстово поле (st.text\_area). Може да бъде въведена ръчно или заредена от предварително дефинираните шаблони (само ако е избран Wikidata endpoint).
  + **SPARQL Endpoint URL:** URL адрес на крайната точка, към която ще бъде изпратена заявката. Потребителят избира от списък с предварително дефинирани endpoints (Wikidata, Europeana) или въвежда собствен URL в текстовото поле (st.text\_input).
  + **Wikidata Търсене:**
    - Тип на ентитите: Избор между "Клас (QID)" или "Предикат (PID)" чрез радио бутони (st.radio).
    - Термин за търсене: Текст, въведен от потребителя в поле за търсене (st.text\_input).
  + **Потребителски действия:** Кликане върху бутони ("ИЗПЪЛНИ ЗАЯВКА", "Копирай код", "Изчисти редактора") и избор от менюта (избор на endpoint, избор на шаблон).
* **Обработка:**
  + **Управление на състоянието:** При всяко взаимодействие, Streamlit запазва текущото състояние (въведена заявка, избран endpoint, резултати от търсене и др.) в st.session\_state, за да осигури последователност между повторните изпълнения на скрипта.
  + **Търсене на Wikidata Ентитита:**
    - Функцията search\_wikidata\_entities (от sparql\_utils.py) изпраща HTTP GET заявка към Wikidata API (wbsearchentities) с въведения термин и избрания тип.
    - API отговорът (JSON) се парсва.
    - Резултатите се форматират (етикет, ID, описание) и се кешират (@st.cache\_data) за подобряване на производителността при повторни търсения.
    - Обработват се евентуални грешки при комуникацията с API.
  + **Изпълнение на SPARQL Заявка (при натискане на бутона "ИЗПЪЛНИ ЗАЯВКА"):**
    - Функцията execute\_sparql\_query (от sparql\_utils.py) получава текущата заявка и избрания endpoint URL.
    - Формира се HTTP GET заявка към endpoint-а, като SPARQL заявката се предава като URL параметър.
    - Изпраща се заявката чрез библиотеката requests.
    - Отговорът от endpoint-а се обработва според неговия Content-Type:
      * application/sparql-results+json: JSON отговорът се парсва. Ако съдържа results.bindings (за SELECT заявки), те се преобразуват в pandas.DataFrame. Ако съдържа boolean (за ASK заявки), булевата стойност се извлича.
      * application/sparql-results+xml, text/csv: Връща се суровият текстов отговор и се прави опит за парсване на CSV в DataFrame, ако е възможно.
    - Обработват се различни типове грешки (Timeout, HTTP грешки, грешки при парсване на JSON/XML/CSV) и се генерира съобщение за грешка.
* **Изход и използване:**
  + **Резултати от SPARQL Заявка:**
    - При успешен SELECT: Резултатите се показват в интерактивна таблица (st.dataframe) в дясната колона на интерфейса.
    - При успешен ASK: Показва се булевият резултат (True/False) в дясната колона.
    - Суров Отговор: Суровият JSON отговор от endpoint-а се показва в разгъваема секция (st.expander) под основните резултати.
    - Съобщение за Грешка: Ако възникне грешка при изпълнението, се показва това съобщение за грешка (st.error) в дясната колона.
  + **Резултати от Wikidata Търсене:**
    - Списък на намерените ентитита (етикет, ID, описание) се показва динамично в страничната лента.
    - Всеки резултат има бутон "Копирай код", който чрез библиотеката pyperclip копира ID-то на ентитите (с префикс wd: или wdt:) в клипборда на потребителя, за лесно поставяне в редактора на заявки.
  + **Обратна връзка към потребителя:** Информационни съобщения (st.info, st.warning) показват състоянието в което се намираме след подадената заявка - празна заявка, успешно изпълнение без резултати, зареждане на шаблон и др.

## Тип и представяне на съдържанието

Проектът работи със следните източници на данни и метаданни:

* **SPARQL Endpoints:** Това са основните източници на данни. Приложението е предварително конфигурирано с URL адресите на популярни публични endpoints като:
  + Wikidata SPARQL Endpoint (https://query.wikidata.org/sparql)
  + Europeana SPARQL Endpoint (https://sparql.europeana.eu/)  
    Потребителите могат да въвеждат и URL адреси на други SPARQL endpoints. Тези endpoints предоставят достъп до големи графи от свързани данни (RDF), които могат да бъдат заявявани чрез SPARQL.
* **Wikidata API (wbsearchentities):** Използва се като източник на метаданни за търсене на Wikidata ентитита (класове и предикати) по техните етикети или описания. Този API връща информация като ID (QID/PID), етикет и описание за съответстващите термини.
* **Вътрешни шаблони за заявки:** Файлът query\_templates.py съдържа речник с текстови SPARQL заявки, специфични за Wikidata, които служат като примери и отправни точки за потребителя. Примерите са с различна сложност.

**Тип на съдържанието:**

* **Текстово съдържание:**
  + SPARQL заявки (въвеждани от потребителя или зареждани от шаблони).
  + URL адреси на endpoints.
  + Термини за търсене в Wikidata.
  + Етикети, ID-та и описания на Wikidata ентитита (получени от API).
  + Резултати от SPARQL заявки (представени като текст, таблици или суров JSON/XML/CSV).
  + Съобщения за грешки и информационни съобщения в интерфейса.
* **Структурирани данни:**
  + JSON: Основният формат за отговор от SPARQL и от Wikidata API. Приложението парсва този формат за извличане на резултати и метаданни.
  + XML/CSV: Алтернативни формати за отговор от SPARQL endpoints, които приложението може да получи и покаже като суров текст (с опит за парсване на CSV в таблица).
* **Графично/Аудио/Видео съдържание:** Проектът **не** обработва и **не** представя директно графично, аудио или видео съдържание. Резултатите от заявките могат да съдържат URL-и към такива ресурси, но самото приложение не ги визуализира или възпроизвежда.

**Ресурси, размер и кодиране:**

* **Текстови документи/ресурси:**
  + Основният код на приложението (app.py, sparql\_utils.py, query\_templates.py) – няколко Python файла, общ размер под 1 MB, кодирани в UTF-8.
* **Външни ресурси (API/Endpoints):**
  + Достъпват се чрез HTTP GET заявки.
  + Размерът на отговорите варира значително в зависимост от сложността на SPARQL заявката и размера на множеството от резултати (от няколко байта за ASK заявка, до мегабайти за големи SELECT резултати).
  + Типът на файловете/отговорите е предимно JSON, но може да бъде и XML или CSV.
  + Кодирането на отговорите обикновено е UTF-8, което се обработва стандартно от библиотеките requests и pandas, които използваме.

## Използване на изкуствен интелект (ИИ) за разработването на проекта

При разработването на проекта беше използван изкуствен интелект (ИИ) (Github Copilot) като помощно средство в следните аспекти:

1. **Насоки и структура:** ИИ беше използван за генериране на първоначални идеи за структурата на Streamlit приложението и възможните потребителски интерфейсни елементи.
2. **Кодови фрагменти и примери:** ИИ помогна за генерирането на стандартни кодови фрагменти за изпращане на HTTP заявки с библиотеката requests и за основна обработка на грешки. Също така предостави примери за използване на st.session\_state в Streamlit.
3. **Ресурси и препратки:**  ИИ предложи използването на библиотеката pyperclip за функционалността за копиране в клипборда и предостави връзки към нейната документация.
4. **Отстраняване на неизправности и разрешаване на проблеми:**  При възникване на специфична грешка при парсване на JSON отговори от определен SPARQL endpoint, ИИ помогна да се идентифицира потенциалната причина и предложи начини за по-устойчива обработка на данните.
5. **Обратна връзка и итерация:**  ИИ беше използван за преглед на части от кода за потенциални подобрения в ефективността.
6. **Документиране на проекта:**  ИИ беше използван за генериране на първоначални чернови на описанията на функциите в sparql\_utils.py.

Степента на използване на ИИ беше основно като **помощник и източник на информация и примери**, като крайните решения за архитектурата, имплементацията и съдържанието на документацията бяха взети от нас като автори. Генерираният от ИИ код беше прегледан, тестван и адаптиран към идеята и визията ни за проекта.

# Дизайн

Тази секция представя дизайна на уеб приложението SPARQL Query Builder, което решава проблема с улесняването на работата със SPARQL заявки.

**Дизайн на решението и използване на технологии:**

Решението е реализирано като едностранично уеб приложение с помощта на Python и библиотеката Streamlit. Изборът на Streamlit е ключов, тъй като позволява бързо прототипиране и разработка на интерактивни уеб интерфейси директно от Python код, без нужда от писане на отделен frontend код от HTML, CSS или JavaScript.

* **Streamlit (app.py):** Използва се за:
  + Дефиниране на структурата на потребителския интерфейс (странична лента, две колони).
  + Създаване на UI компоненти (текстови полета, бутони, dropdown менюта, радио бутони, таблици, разгъваеми секции).
  + Обработка на потребителски вход и събития (промяна на текст, натискане на бутон).
  + Управление на състоянието на приложението между взаимодействията чрез st.session\_state.
  + Визуализация на резултатите и съобщенията за грешки.
* **Requests (sparql\_utils.py):** Библиотеката requests се използва за комуникация с външни HTTP услуги:
  + Изпращане на SPARQL заявки (като GET параметри) към избраните SPARQL endpoints.
  + Изпращане на заявки към Wikidata API за търсене на ентитита.
  + Обработка на HTTP отговорите и статус кодовете.
* **Pandas (sparql\_utils.py, app.py):** Използва се за:
  + Преобразуване на JSON резултатите от SPARQL SELECT заявки (results.bindings) в структуриран DataFrame.
  + Удобно представяне на табличните резултати в Streamlit интерфейса чрез st.dataframe.
* **Pyperclip (app.py):** Използва се за реализиране на функционалността "Копирай код" в резултатите от търсенето в Wikidata, като копира ID-то на ентитите в системния клипборд.
* **Структуриране на кода:** Логиката е разделена на модули за по-добра организация:
  + app.py: Основен файл, UI логика и управление на процесите.
  + sparql\_utils.py: Помощни функции за работа със SPARQL и Wikidata API.
  + query\_templates.py: Съхранение на шаблони за заявки.

**Архитектура на решението:**

Приложението следва проста клиент-сървър архитектура, където Streamlit сървърът изпълнява Python кода и рендира HTML/JavaScript интерфейса в браузъра на потребителя. От гледна точка на данните, приложението действа като клиент към външни услуги (SPARQL endpoints, Wikidata API).

**Схема на взаимодействие:**

1. **Потребител:** Взаимодейства с уеб интерфейса в браузъра си (въвежда заявка, избира endpoint, търси ентитита, натиска бутони).
2. **Браузър:** Изпраща действията на потребителя към Streamlit сървъра.
3. **Streamlit Сървър (app.py):**
   * Получава взаимодействието.
   * Извиква съответните функции от sparql\_utils.py (за изпълнение на заявка или търсене).
   * Управлява st.session\_state.
   * Подготвя данните за показване.
   * Рендира обновения HTML/JS интерфейс и го изпраща обратно към браузъра.
4. **sparql\_utils.py:**
   * При извикване, използва библиотеката requests за изпращане на HTTP заявки към:
     + **Външен SPARQL Endpoint:** (напр. Wikidata, Europeana).
     + **Wikidata API:** (wbsearchentities).
   * Получава HTTP отговорите (JSON, XML, CSV).
   * Парсва и обработва отговорите (използвайки json, pandas).
   * Връща резултатите или грешките към app.py.
5. **query\_templates.py:** Предоставя статичен речник с шаблони към app.py, когато е необходимо.

**Проектиране (стъпки и итерации):**

Проектирането протече в следните стъпки:

1. **Първоначална концепция:** Дефиниране на основните изисквания – редактор на заявки, избор на endpoint, изпълнение, показване на резултати.
2. **Избор на технология:** Избран беше Streamlit поради бързината на разработка и лесната интеграция с Python (Pandas, Requests).
3. **Основна функционалност:** Имплементиране на ядрото – текстово поле за заявка, избор на endpoint, бутон за изпълнение, извикване на execute\_sparql\_query и показване на отговора.
4. **Подобряване на визуализацията:** Интегриране на Pandas за таблично представяне на SELECT резултатите (st.dataframe).
5. **Интеграция с Wikidata:** Добавяне на шаблони за Wikidata и функционалност за търсене на ентитита (search\_wikidata\_entities, интеграция с Wikidata API, кеширане).
6. **Потребителски интерфейс и изживяване:** Организиране на интерфейса в колони и странична лента, добавяне на бутони за копиране и изчистване, подобряване на обратната връзка чрез информационни съобщения и обработка на грешки.
7. **Управление на състоянието:** Имплементиране на st.session\_state за запазване на контекста между действията на потребителя.

**Примерен код и използване на ресурси:**

*Примерен код от sparql\_utils.py (изпълнение на заявка):*

import requests

import pandas as pd

import streamlit as st

import json

# @st.cache\_data

def execute\_sparql\_query(endpoint\_url, query):

    headers = {

        'Accept': 'application/sparql-results+json, application/json',

        'User-Agent': 'SPARQLQueryBuilderApp/1.0 (Python-Requests)'

    }

    params = {'query': query, 'format': 'json'} # Изискваме JSON по подразбиране

    try:

        response = requests.get(endpoint\_url, headers=headers, params=params, timeout=30)

        response.raise\_for\_status() # Хвърля HTTPError за лоши отговори

        content\_type = response.headers.get('Content-Type', '').split(';')[0]

        if 'application/sparql-results+json' in content\_type or 'application/json' in content\_type:

            results\_json = response.json()

            if 'results' in results\_json and 'bindings' in results\_json['results']:

                bindings = results\_json['results']['bindings']

                if not bindings:

                    return None, results\_json, None # Няма резултати, но заявката е успешна

                data = [{key: value['value'] for key, value in item.items()} for item in bindings]

                df = pd.DataFrame(data)

                return df, results\_json, None

            # Обработка на ASK резултати

            elif 'boolean' in results\_json:

                return results\_json['boolean'], results\_json, None

            else:

                 return None, results\_json, f"Received JSON, but format not recognized: {list(results\_json.keys())}"

        elif 'application/sparql-results+xml' in content\_type or 'text/csv' in content\_type:

             # Връщаме текста за други формати

             return None, response.text, f"Received non-JSON format: {content\_type}"

        else:

           return None, response.text, f"Unsupported Content-Type: {content\_type}"

    except requests.exceptions.Timeout:

        return None, None, "Error: The request timed out."

    except requests.exceptions.HTTPError as http\_err:

        return None, response.text if response else None, f"HTTP Error: {http\_err}"

    except requests.exceptions.RequestException as req\_err:

        return None, None, f"Request Error: {req\_err}"

    except json.JSONDecodeError:

         return None, response.text, "Error: Could not decode JSON response."

    except Exception as e:

        return None, None, f"An unexpected error occurred: {e}"

*Обяснение:* Тази функция приема URL на endpoint и SPARQL заявка. Тя задава подходящи HTTP хедъри, изпраща GET заявка с requests, и обработва отговора. Ако отговорът е успешен и е в JSON формат, тя парсва JSON-а. За SELECT заявки, резултатите се преобразуват в Pandas DataFrame. За ASK заявки, булевата стойност се връща директно. Ако форматът е друг (XML, CSV) или възникне грешка (timeout, HTTP грешка, грешка при парсване), функцията връща съответно суровия отговор или съобщение за грешка, заедно с None за DataFrame/boolean резултата.

*Примерен код от app.py (използване на st.session\_state и UI елемент):*

import streamlit as st

if 'current\_query\_text' not in st.session\_state:

    st.session\_state.current\_query\_text = "" # Default query

if 'results\_df' not in st.session\_state:

    st.session\_state.results\_df = None

query\_input = st.text\_area(

    "SPARQL Query",

    value=st.session\_state.current\_query\_text,

    height=300,

    key="query\_text\_area\_key",

    on\_change=handle\_query\_text\_change

)

def handle\_query\_text\_change():

    # Нулиране на резултатите, когато заявката се промени ръчно

    if st.session\_state.current\_query\_text != st.session\_state.query\_text\_area\_key:

        st.session\_state.results\_df = None

        st.session\_state.raw\_results\_response = None

        st.session\_state.query\_error\_message = None

        st.session\_state.last\_loaded\_template\_name = None

    st.session\_state.current\_query\_text = st.session\_state.query\_text\_area\_key

*Обяснение:* Този фрагмент показва как st.session\_state се използва за съхраняване на текущия текст на заявката (current\_query\_text). UI елементът st.text\_area показва тази стойност. Когато потребителят промени текста, функцията handle\_query\_text\_change се извиква автоматично (чрез on\_change). Тази функция актуализира st.session\_state.current\_query\_text с новата стойност от текстовото поле (st.session\_state.query\_text\_area\_key) и нулира предишните резултати, за да се избегне показването на стари данни с нова заявка.

# Тестване

Тестването на приложението SPARQL Query Builder беше проведено с цел да се гарантира неговата функционалност, стабилност и коректност при работа с различни входни данни и сценарии.

* **С колко и какви данни:**
  + **SPARQL Заявки:** Тествани бяха разнообразни SPARQL заявки:
    - Прости SELECT заявки с различен брой променливи и LIMIT клаузи.
    - SELECT заявки с FILTER, OPTIONAL, GROUP BY, ORDER BY.
    - ASK заявки (очакващи True или False).
    - CONSTRUCT и DESCRIBE заявки (очакващи RDF формати - тествано е дали приложението ги изпълнява без грешка и показва суровия отговор).
    - Заявки, връщащи голям брой резултати (за тестване на производителността на st.dataframe).
    - Заявки, не връщащи резултати.
    - Синтактично некоректни SPARQL заявки (за тестване на обработката на грешки от страна на endpoint-а).
    - Заявки от вградените Wikidata шаблони.
  + **SPARQL Endpoints:**
    - Основно тестване срещу **Wikidata SPARQL Endpoint**: Използвани са множество заявки, включително тези от шаблоните и специфични за Wikidata структурата (използване на wdt:, wd: префикси).
    - Тестване срещу **Europeana SPARQL Endpoint**: Изпълнени са няколко примерни заявки, за да се провери съвместимостта с друг публичен endpoint.
    - Тестване с **невалидни/недостъпни URL-и** за потребителски endpoint (за проверка на обработката на мрежови грешки и timeout).
  + **Wikidata Търсене:**
    - Търсене на често срещани класове (напр. "human", "city") и предикати (напр. "instance of", "population").
    - Търсене на по-редки или специфични термини.
    - Търсене с термини на кирилица и латиница (където API поддържа).
    - Търсене на термини, които не съществуват (за проверка на празен резултат).
    - Търсене с празен низ.
* **В какви среди е тестването:**
  + **Локална машина:** Основното тестване беше проведено на локална машина под Windows 11.
  + **Python Версия:** Python 3.13.
  + **Библиотеки:** Версиите на Streamlit, Pandas, Requests, Pyperclip, инсталирани по време на разработката – запазени са версиите в requirements.txt файла в репото в което си споделяхме кода.
  + **Браузъри:** Тествано е основно с Google Chrome, работи и на Microsoft Edge.
* **Какво сте получили като резултат:**

Тестовете показаха, че приложението успешно изпълнява основните си функции:

* + **Изпълнение на заявки:** Коректно изпраща заявки към избраните endpoints и получава отговори.
  + **Показване на резултати:**
    - SELECT резултатите се визуализират правилно в st.dataframe.
    - ASK резултатите се показват като True или False.
    - Суровите JSON отговори се показват коректно в разгъваемата секция.
    - При заявки, връщащи други формати (XML/CSV), се показва суровият текстов отговор.
  + **Wikidata Интеграция:**
    - Шаблоните за Wikidata се зареждат и изпълняват успешно.
    - Търсенето на Wikidata ентитита работи според очакванията, връщайки релевантни резултати.
    - Функцията "Копирай код" успешно копира ID-то в клипборда.
    - Кеширането на резултатите от търсенето подобрява бързодействието при повторни търсения.
  + **Обработка на грешки:**
    - Грешки от SPARQL endpoint (напр. синтактични грешки в заявката) се улавят и показват на потребителя заедно със съобщението от сървъра.
    - Мрежови грешки (невалиден endpoint, timeout) се обработват и се показват ясни съобщения.
    - Грешки при парсване на неочаквани формати също се индикират.
  + **Управление на състоянието:** st.session\_state успешно запазва въведената заявка, избрания endpoint и последните резултати/грешки при навигация и взаимодействие с UI елементите.
  + **Екранна снимка 1:** Показва интерфейса след успешно изпълнение на SELECT заявка към Wikidata, с резултати в таблица и видима секция със суров JSON отговор.  
    A screenshot of a computer

    AI-generated content may be incorrect.
  + **Екранна снимка 2:** Демонстрира резултати от търсене на Wikidata ентитите в страничната лента с бутони "Копирай код".  
    A screenshot of a phone

    AI-generated content may be incorrect.
  + **Екранна снимка 3:** Показва съобщение за грешка, получено при опит за изпълнение на синтактично некоректна SPARQL заявка.  
    A screenshot of a computer

    AI-generated content may be incorrect.

# Заключение и възможно бъдещо развитие

Макар и проекта да е функционален и да покрива основните си цели, съществуват възможни подобрения:

* **Разширен Query Builder**: Интегриране на по-визуални или структурирани начини за изграждане на заявки, например чрез избор на предикати и обекти от списъци, вместо ръчно писане.

* **Syntax Highlighting**: Добавяне на оцветяване на синтаксиса в текстовото поле за SPARQL заявки за по-добра четимост.
* **Валидация на SPARQL заявки**: Клиентска валидация на синтаксиса на заявката преди изпращането й към сървъра.
* **Експорт на резултати**: Възможност за експорт на резултатите в различни формати (CSV, JSON, Excel).
* **История на заявките**: Запазване на история на изпълнените заявки в рамките на сесията или дори локално.

# Разпределение на работата

Работата по проекта беше разпределена между двамата автори, както следва:

* **Дарин Крумов (0MI0200003):**
  + Проектиране на основната архитектура на приложението.
  + Разработване на основната логика на Streamlit приложението (app.py), включително структурата на UI.
  + Стилизиране и настройване на потребителския интерфейс.
  + Подготовка на част от документацията (секции Дизайн, Тестване).
  + Провеждане на част от тестовете.
* **Самуил Ганев (6MI0200001):**
  + Разработване на модула за работа със SPARQL заявки (execute\_sparql\_query в sparql\_utils.py), включително обработката на различни формати на отговор и управлението на грешки.
  + Създаване на шаблоните за Wikidata заявки (query\_templates.py).
  + Имплементиране на функционалността за търсене на Wikidata ентитита и интеграция с Wikidata API (search\_wikidata\_entities в sparql\_utils.py).
  + Имплементиране на функционалността за избор на endpoint и зареждане на шаблони в интерфейса.
  + Интегриране на библиотеката pyperclip за копиране в клипборда.
  + Подготовка на част от документацията (секции Въведение, Анализ на решението, Заключение).
  + Провеждане на част от тестовете и събиране на екранни снимки.

Разработването се извършваше съвместно, с регулярна комуникация и споделяне на код чрез Git. И двамата участваха в обсъждането на дизайна, решаването на проблеми и финалното тестване и документиране.

# Използвани литературни източници и Уеб сайтове

1. **Streamlit Documentation.** [https://docs.streamlit.io/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdocs.streamlit.io%2F)
2. **Requests: HTTP for Humans.** [https://requests.readthedocs.io/en/latest/](https://www.google.com/url?sa=E&q=https%3A%2F%2Frequests.readthedocs.io%2Fen%2Flatest%2F)
3. **Wikidata Query Service User Manual.** [https://www.wikidata.org/wiki/Wikidata:SPARQL\_query\_service/User\_Manual](https://www.google.com/url?sa=E&q=https%3A%2F%2Fwww.wikidata.org%2Fwiki%2FWikidata%3ASPARQL_query_service%2FUser_Manual)
4. **Wikidata API Help.** [https://www.wikidata.org/w/api.php?action=help&modules=wbsearchentities](https://www.google.com/url?sa=E&q=https%3A%2F%2Fwww.wikidata.org%2Fw%2Fapi.php%3Faction%3Dhelp%26modules%3Dwbsearchentities)
5. **SPARQL 1.1 Query Language.** W3C Recommendation. [https://www.w3.org/TR/sparql11-query/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fwww.w3.org%2FTR%2Fsparql11-query%2F)
6. **Europeana SPARQL endpoint.** [https://sparql.europeana.eu/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fsparql.europeana.eu%2F)

# Апендикс

Няма допълнителни материали в апендикса.