UNIVERSITATEA „ALEXANDRU IOAN CUZA” DIN IAŞI
FACULTATEA DE ECONOMIE ȘI ADMINISTRAREA AFACERILOR
SPECIALIZAREA: SISTEME INFORMAȚIONALE PENTRU AFACERI

LUCRARE DE DIZERTAȚIE

DEZVOLTAREA DE APLICAȚII ÎN PYTHON FOLOSIND BAZE DE DATE DE TIP GRAF

Profesor Coordonator,
Prof. Dumitriu Florin

Student,
Cojocariu Daniel

IAȘI
Iunie, 2025

CUPRINS

INTRODUCERE

Tema acestei lucrări constă în dezvoltarea unei aplicații folosind limbajul de programare Python și o bază de date de tip graf. Această arhitectură este din ce în ce mai folosită datorită numărului mare de instrumente disponibile pentru a face posibilă această integrare și limbajul python fiind unul popular, simplu și ușor de înțeles, poate fi utilizat indiferent de experiența unui programator. În ultimii ani, tehnologia a evoluat foarte rapid în domeniul bazelor de date de tip graf care la rândul lor au avut un impact major în modelarea relațiilor complexe în procesele business, rețelele sociale sau algoritmii de recomandare. Implicarea inteligenței artificiale în baze de date de tip graf este un alt motiv de popularitate în momentul de față datorită gradului ridicat de utilizare al AI-ului în ultimii ani, unde conceptul de *inteligență artificială* este întâlnit pe majoritatea platformelor și în companiile mari precum Microsoft sau Google.

Obiectivul acestei lucrări este dezvoltarea unei aplicații de tip rețea socială cu un set de funcționalități pentru a livra un prototip funcțional în care utilizatorii utilizează o interfață grafică pentru a interacționa cu alți utilizatori. Aplicația va utiliza o arhitectură standard, o bază de date de tip graf și limbajul de programare **Python** care va avea rolul de **backend** și **frontend** folosind diverse instrumente oferite de acest limbaj.

Lucrarea de față este structurată pe trei capitole în care sunt prezentate informațiile cheie despre bazele de date de tip graf, limbajul de programare **python** și aplicații sociale.

În primele două capitole lucrarea de față va aprofunda despre concepte generale, instrumente folosite, limbajul de interogare **Cypher** pentru platforma **Neo4j**, avantaje și dezavantaje pentru baze de date de tip graf. Al doilea capitol explică limbajul de programare **Python** și interacțiunea acestui limbaj cu **bazele de date de tip graf** în contextul unei aplicații de tip rețea socială.

Dezvoltarea aplicației pentru această lucrare este explicată la capitolul trei unde este utilizată platforma **Neo4j**, una dintre cele mai populare soluții pe piață pentru baze de date de tip graf împreună cu limbajul de programare **Python**. Capitolul trei va conține diverse bucăți de cod, cum ar fi scripturi de crearea a bazei de date, logica de accesare a unui utilizator și implementarea unui algoritm de recomandare care calculează un scor pentru a recomanda postări în funcție de interacțiunile utilizatorului cu aplicația. Pe lângă aceste concepte lucrarea va conține exemple de proiectare a anumitor interfețe utilizator, a structurii codului și alte concepte generale.

CAPITOLUL 1. Concepte generale privind bazele de date de tip graf

Înainte de a discuta din punct de vedere tehnic despre bazele de date de tip graf, voi prezenta câteva detalii generale generale ale bazelor de date. Acestea au apărut în diverse forme, însă obiectivul unei baze de date este de a organiza informații. O bază de date oferă posibilitatea de a interacționa cu informații folosind un limbaj predefinit dictat de platforma folosită. Aceste interacțiuni pot fi categorizate în patru secțiuni (Kemper, 2015):

- **Data definition**: Orice definiție care modifică organizarea datelor într-o bază de date
- **Update**: O acțiune care manipulează informații dintr-o baza de date
- **Retrieval**: Informațiile sunt stocate într-o bază de date și pot fi accesate de o aplicație sau sistem
- **Administration**: Constă în acțiunile făcute de persoane calificate, acțiuni precum: securitate sau analiza performanței

În prezent există două tipuri populare de baze de date:

a) **Baze de date relaționale** *(SQL[1])*, cum ar fi **PostgreSQL** sau **MySQL**. Stochează informații în tabele unde coloanele reprezintă atribute și rândurile fiecare entitate. Când sunt necesare interogări complexe sunt folosite tehnici de combinare a tabelelor de exemplu *join sau union.*

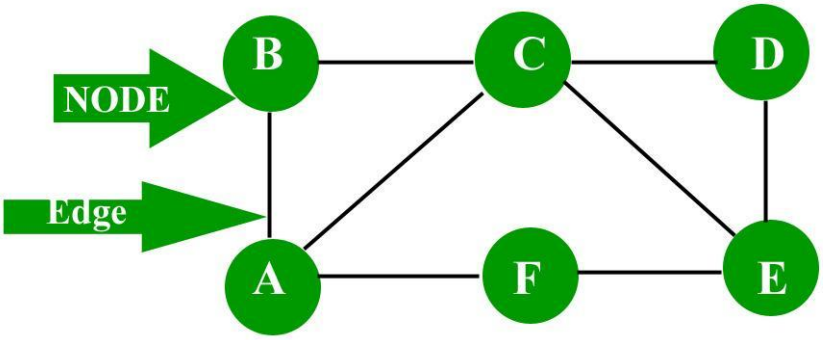
b) **Baze de date** *NoSQL*: Aceste tipuri de baze de date apar în diverse forme cum ar fi Key-values, IntraDB sau Document-oriented.

1.1. Baze de tip graf - limbaje și instrumente

Prima lucrare cunoscută despre teoria grafurilor a fost scrisă în 1736, intitulată „*Șapte poduri de la Königsberg*", de Leonhard Euler, un matematician și fizician strălucit, considerat a fi matematicianul remarcabil al secolului al **XVIII-lea**. El a introdus o mare parte din notația și terminologia utilizată în matematica modernă și a publicat pe larg în domeniile dinamicii fluidelor, astronomiei și chiar teoriei muzicii. Leonhard Euler a contribuit la dezvoltarea matematicii moderne și a altor domenii până unde sunt astăzi (Kemper, 2015). Înainte de a discuta în detaliu cum funcționează o bază de date de tip graf, trebuie să definim ce este un graf.

Un graf este un obiect matematic definit prin *noduri* și *muchii.*

[1] Structured Query Language (limbaj structurat de interogare)



Figură 1 Noduri și muchii într-un graf matematic (Mathematics | Graph Theory Basics - Set 1, 2025)

Folosind un exemplu din realitate, un graf poate fi reprezentat de o hartă, unde nodurile sunt orașe și muchiile drumurile[2].

Grafurile pot fi aplicate și folosite în mai multe domenii sau sisteme cum ar fi:

a) Rețele sociale: Fiecare utilizator este reprezentat de un nod, iar legăturile sau relațiile dintre utilizatori cu alte obiecte sunt legate prin muchii.
b) Algoritmi de recomandare: Grafurile sunt utilizate destul de des pentru a recomanda produse sau conexiuni între utilizatori.
c) Sisteme de antifraudă: În domeniul financiar sau cybersecurity o bază de date de tip graf ajută la detectarea diverselor anomalii.

1.1.1. Limbaje utilizate în baze de date

Există diverse limbaje utilizate în funcție de baza de date folosită. Indiferent de limbaj, acesta oferă o modalitate de a interacționa cu o bază de date care permite manipularea, definirea și interogarea datelor. Cel mai cunoscut și utilizat limbaj este SQL (Strucured Query Language), folosit de baze de date relaționale (PostgreSQL, SQL Server, Oracle) . Cele mai comune operații folosite într-un limbaj SQL sunt: *create, select, update, insert* și *delete*.

În această lucrare, s-a utilizat limbajul **Cypher** oferit de platforma **Neo4j,** definiția acestui limbaj este regăsită în documentația *Cypher Query Language* fiind următoarea "*Cypher este un limbaj declarativ de interogare grafică care permite interogarea expresivă, eficientă și actualizarea unei baze de date de tip graf* "(Neo4j I. , 2021).

[2] https://www.geeksforgeeks.org/mathematics-graph-theory-basics-set-1/

1.1.2. Instrumente

Instrumentele folosite de baze de date variază de la o platformă la alta. Bazele de date oferă instrumente, framework-uri și medii de dezvoltare.

a) Postgres utilizează aplicația software **pgAdmin**, o interfață grafică care permite gestionarea bazelor de date și monitorizarea performanței.

b) Oracle utilizează aplicația **Oracle SQL Developer,** pentru gestionarea, scrierea și executarea de interogări SQL. Un alt instrument oferit este **Oracle Enterpise Manager** care oferă funcționalități de monitorizare a performanței și configurare a instanțelor oracle.

c) MongoDB oferă aplicația software **MongoDB Compass**, o interfață grafică pentru vizualizarea și administrarea datelor, analizarea performanței unei baze de date. O altă aplicație este **MongoDB Atlas**, o platformă cloud oficială specifică pentru securitate, monitorizare și deployment

d) MySQL oferă diverse instrumente cum ar fi MySQL Workbench, MySQL Shell și MySQL Enterprise Backup.

e) Microsoft SQL Server oferă instrumente oficiale precum **SQL Server Management** (SSMS) care permite rularea, gestionarea și administrarea unei baze de date. Un alt instrument folosit de aceasta platformă este **Azure Data Studio**, folosită pentru a accesa o bază de date cu extensii externe precum: Jupyter Notebooks și Git Integration.

f) Bazele de tip graf utilizează diverse instrumente pentru a gestiona o bază de date, de exemplu Neo4j Desktop și Neo4j Browser, aceste aplicații oferă o interfață grafică pentru administrarea și configurarea bazelor de date de tip graf unde este folosit limbajul **Cypher**.

1.1.3. Tipuri de interogări în baze de date de tip graf

Există diferite tipuri și tehnici utilizate pentru a crea diverse interogări, cel mai simplu exemplu de interogare este una de tip "**single-hop query**" care ar putea fi reprezentată de următoarea întrebare "Care sunt toți prietenii utilizatorului X?". Folosind limbajul Cypher pe platformă Neo4j, acest rezultat poate fi realizat prin interogarea:

„MATCH(u:Utilizator {name: ‚X'}) RETURN u

Cele mai des utilizate interogări într-o bază de tip graf sunt următoarele:

- **Single Hop**: Cum am explicat anterior, această interogare este una directă și simplă care returnează un set de date în funcție de anumite condiții.
- **Multi Hop**: Această interogare este mai complexă, față de *Single Hop* utilizează mai mulți pași pentru returna un set de date de exemplu: "*Care sunt prieteni prietenilor utilizatorului X*". Într-o bază de date relațională pentru a realiza acest tip de interogare se folosesc mai multe *JOIN-uri.*
- **Pattern-matching query:** Această interogare ajută la găsirea de relații între noduri care respectă anumite criterii: „Care sunt utilizatorii care au achiziționat același produs"
- **Shortest path queries:** Interogările acestea caută cea mai scurtă cale de la un nod la altul, fiind util în contextul de logistică și hărți.
- **Aggregation queries:** Interogările agregate se referă la returnea de date calculate folosind diverse funcții precum: count, sum, average.

1.2. Limbajul Cypher

Cypher este un limbaj de interogare specializat pentru baze de date de tip graf și reprezintă principalul instrument pentru a interacționa cu acest tip de informații pe platforma Neo4j.

Acest limbaj a fost creat cu scopul de a fi expresiv, intuitiv și ușor de înțeles fiind foarte asemănător cu un limbaj SQL în formularea interogărilor, dar oferind în același timp capabilitățile necesare de a manipula o bază de date de tip graf.

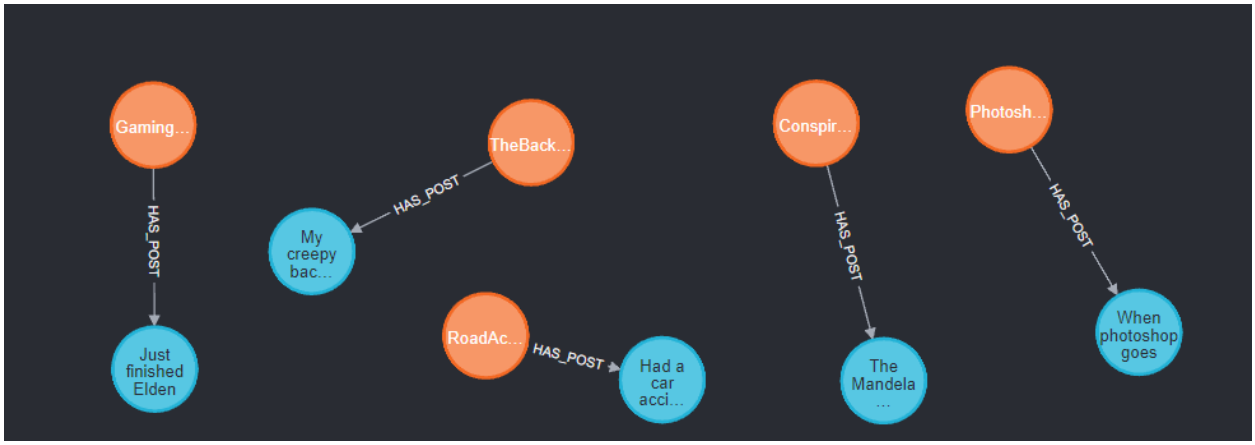
1.2.1. Sintaxa limbajului Cypher

Înterogările Cypher folosesc diverse cuvinte sau *patterns* cum ar fi MATCH, RETURN acestea fiind o inspirație din limbajul SQL care utilizează SELECT, UPDATE etc. Cypher se concentrează pe idea de a fi ușor de înțeles, astfel utilizează diverse semne vizuale precum **()** pentru noduri, -- pentru relații nedirecționale și --> pentru relații direcționale. Următorul exemplu subliniază exact această situație:

```
MATCH(c:Community)-[:HAS_POST]→(p:Post) RETURN c,p;
```

Figură 2 Interogare Cypher în platforma Neo4j Browser

Rezultatul acestei interogări arată în felul următor:



Figură 3 Rezultat interogare Cypher de pe platforma Neo4j Browser (Neo4j I. , 2021)

Această interogare caută toate perechile de Comunități legate printr-o **relație** denumită HAS_POST cu Postări folosind condițiii inspirate din SQL:

- **MATCH** : identifică modele în graf
- **WHERE** : condiție prin specificarea unei relații între semnele "-" și "->"
- **RETURN** : definirea informațiilor care o să fie returnate

Aceste condiții pot fi combinate pentru a crea interogări complexe, inclusiv traversări, multi-hop, agregări, ordonări și filtrări.

1.2.2. Tipuri de date și funcții

Cypher suportă tipuri **standard** de date care pot fi regăsite într-un limbaj **SQL** (string, integer, boolean, list etc), dar și tipuri specifice unei baze de date de tip graf (node, relationship, path etc). Acest limbaj oferă și funcții integrate precum: manipularea șirurilor de caractere (toLower, substring), funcții de listă (size, collect), funcții de agregare (count, sum, avg), funcții pentru navigare în graf (length, shortestPath). Modelele de interogări folosite în general sunt următoarele:

- Traversări simple (eg: Utilizatori direcți)
- Traversări recursive (ex: prieteni ai prietenilor)
- Detectarea de pattern-uri specifice
- Căutarea celei mai scurte căi dintre două noduri.

Cypher fiind un limbaj prietenos și robust pentru gestionarea și analizarea de grafuri, acest limbaj facilitează modelarea și explorarea relațiilor complexe dintre entități pentru diverse modele indiferent de domeniu.

1.3. Avantaje și dezavantaje ale bazelor de date de tip graf

Bazele de tip graf sunt diferite față de o bază de date tradițională prin simplu fapt că pune accent pe relațiile dintre entități și face posibilă crearea de modele foarte complexe.

1.3.1. Avantajele unei baze de date de tip graf

Bazele de date de tip graf sunt flexibile deoarece oferă posibilitatea de a adăuga date, relații și proprietăți fără a afecta schema bazei de date față de o bază de date relațională unde nu se pot adauga date noi cu parametrii diferiți într-o tabelă.

a) Scalabilitate și adaptabilitate: Diverse platforme moderne sunt concepute pentru a fi **scalabile**, permițând popularea unei baze de date de tip graf fără a afecta performanța interogărilor. Utilizarea de **cluster-uri** este un concept popular în cadrul bazelor de date relaționale însă acest concept poate fi utilizat și pentru o bază de date de tip graf care poate beneficia de gestionarea unui volum mare de date și de interacțiuni concurente. Schema flexibilă oferă posibilitatea ca o bază de date de tip graf să crească o dată cu **cerințele** aplicației, menținând performanțele dorite la adăugarea de noi surse de date.

b) Modelare: Datele fiind reprezentate sub formă de graf reflectă în mod intuitiv relațiile din lumea reală facilitând înțelegerea legăturilor dintre noduri (entiăți). Prin interogarea unui graf se pot folosi diverse interogări pentru a descoperii modele și informații legat de interacțiunea dintre entități cum ar fi legăturile utilizatorilor în anumite scenarii. La o bază de date tradițională, depistarea acestor legături este mult mai dificilă și costisitoare.

c) Flexibilitate: Natura unei baze de date de tip graf în esență este flexibilă, permite adăugarea de noduri și relații fără a restructura schema curentă sau a afecta relațiile deja existente.

1.3.2. Dezavantajele unei baze de date de tip graf

a) **Complexitate**: Gestionarea unei baze de date de tip graf în comparație cu una tradițională reprezintă un dezavantaj în contextul dezvoltării, implementării și mentenanța unei baze de tip graf.

b) **Resurse și costuri**: Fiind o bază de date de tip graf care suportă interogări de tip traversare a unui graf, resursele hardware sunt mai costisitoare, necesită o putere computațională ridicată și resurse de stocare mai mari.

c) **Limbaj de interogare standard**: Una dintre cele mai mari dezavantajele ale unei baze de date de tip graf este limbajul de interogare, bazele de date tradiționale folosesc limbajul **SQL** care a fost dezvoltat de la apariția bazelor de date și este un limbaj standard folosit de diverse opțiuni existente pe piață unde la o bază de tip graf, diferiți furnizori oferă propriul limbaj ceea ce a fragmentat ecosistemul din care fac parte aceste baze de date. Programatori sunt nevoiți să învețe un anumit tip de limbaj pentru fiecare soluție care utilizează baze de tip graf.

d) **Aplicabilitate**: Bazele de date nu reprezintă o soluție optimă în contextul unei baze de date care utilizează un set de date simplu care nu conține relații complexe între entități și care utilizează interogări simple. Această problemă este subliniată prin faptul că bazele de date de tip graf sunt eficiente în cazul în care principala provocare este conectivitatea datelor.

1.4. Soluții existente pe piață

În prezent există diverse opțiuni pentru dezvoltarea și implementare unei baze de tip graf, cele mai populare fiind Amazon Neptune, ArangoDB, Azure Cosmos DB, Google Spanner Graph, Janus Graph, MarkLogic, Neo4j și TigerGraph (Disney, 2023). Fiecare soluție oferă diverse arhitecturi, limbaje de interogare, modele de dezvoltare și obiective. Janus Graph este o soluție optimizată pentru gestionarea a unui număr foarte mare de noduri (zeci de milioane). Mark Logic combină documente, instrumente de filtrare care suportă grafuri semantice.

Rank			DBMS	Database Model	Score		
Jun 2025	May 2025	Jun 2024			Jun 2025	May 2025	Jun 2024
1.	1.	1.	Neo4j	Graph	51.33	+3.42	+6.44
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	22.43	-0.51	-5.28
3.	3.	3.	Aerospike	Multi-model	5.27	-0.19	-0.24
4.	4.	↑ 5.	ArangoDB	Multi-model	2.90	+0.05	-0.37
5.	5.	↓ 4.	Virtuoso	Multi-model	2.71	-0.08	-1.55
6.	6.	6.	OrientDB	Multi-model	2.71	+0.00	-0.54
7.	7.	7.	GraphDB	Multi-model	2.56	-0.03	-0.68
8.	8.	8.	Memgraph	Graph	2.48	-0.06	-0.71
9.	9.	9.	Amazon Neptune	Multi-model	2.14	+0.01	-0.15
10.	↑ 11.	10.	NebulaGraph	Graph	1.89	+0.13	-0.34
11.	↑ 12.	11.	Stardog	Multi-model	1.85	+0.12	-0.23
12.	↓ 10.	12.	JanusGraph	Graph	1.75	-0.03	-0.27
13.	13.	↑ 14.	Fauna	Multi-model	1.43	-0.02	-0.12
14.	14.	↓ 13.	TigerGraph	Graph	1.40	-0.02	-0.40
15.	15.	15.	Dgraph	Graph	1.20	-0.04	-0.33
16.	16.	↑ 18.	SurrealDB	Multi-model	1.07	-0.02	+0.04
17.	17.	↓ 16.	Giraph	Graph	1.05	0.00	-0.12
18.	18.	↑ 19.	Blazegraph	Multi-model	0.76	+0.01	-0.05
19.	19.	↓ 17.	AllegroGraph	Multi-model	0.76	+0.03	-0.37
20.	↑ 21.	↑ 21.	Graph Engine	Multi-model	0.57	0.00	-0.10
21.	↓ 20.	↓ 20.	TypeDB	Multi-model	0.57	-0.03	-0.14
22.	22.	↑ 23.	Fluree	Graph	0.42	-0.03	+0.10

Figură 4 Top-ul bazelor de date de tip graf (https://db-engines.com/en/ranking/graph+dbms)

După cum se poate vedea în figura de mai sus, platforma Neo4j are un scor de popularitate major față de alte baze de date de tip graf, Microsoft Azure Cosmos DB aflându-se la o distanță relativ mare. Neo4j este utilizat în diverse domenii precum:

- Generative AI: Inteligența artificială fiind populară și utilizată în multe domenii, AI-ul și-a făcut simțită prezența și în domeniul bazelor de date de tip graf.
- Analizarea și detectarea fraudei: Neo4j oferă instrumente de detectarea fraudei prin utilizarea a diverse modele pentru identificarea anumitor tip de fraude, detectarea anumitor tipuri de actori care au intenția de a frauda, toate aceste date fiind utilizate ulterior la detectarea fraudelor în viitor.
- Medicină: Grafurile sunt utilizate în medicină prin dezvoltarea de modele complexe care oferă posibilitatea de a analiza și mări procesul de dezvoltare a medicamentelor sau detectarea de boli.

- Lanțuri de aprovizionare: Analizarea lanțului de aprovizionare și eficientizarea acestui proces este posibil prin analizarea produselor, furnizorilor, locațiilor și a relațiilor dintre aceste entități.

CAPITOLUL 2. Aspecte generale privind dezvoltarea aplicațiilor de tip rețea socială folosind Python și baze de date de tip graf

Aplicațiile de tip rețea socială au devenit din ce în ce mai populare în ultimii ani, deoarece oamenii caută noi modalități de a interacționa și partaja experiențe personale. Fie că este o platformă strictă de conversații sau de comunități, aceste aplicații au început să facă parte din viața de zi cu zi. Dezvoltarea unei platforme care sprijină comunicarea între utilizatori și partajarea de conținut pare complicat, mai ales dacă se utilizează o bază de tip graf însă, prin utilizarea corectă a diverse instrumente, tehnologii, acest proces este unul mai prietenos.

Limbajul de programare Python este folosit în această lucrare deoarece este un limbaj simplu, ușor de înțeles și oferă toate instrumentele necesare pentru integrarea unei baze de date de tip graf și dezvoltarea unei aplicații folosind acest tip de bază de date.

2.1. Limbajul de programare Python

„*Python este un limbaj de programare open-source, interpretat, orientat pe obiecte, având o semantică înaltă, este foarte atractiv pentru* ***Dezvoltarea Rapida de Aplicatii (RAD),*** *însă este utilizat și pentru dezvoltarea de scripturi*” (Python Documentation, 2001). Acest limbaj a devenit popular datorită nivelului ridicat de productivitate pe care îl oferă, nu este nevoie de compilarea codului pentru rulare, procesul de *edit-test-debug* este foarte rapid.

Acest limbaj de programare se remarcă prin simplul fapt că originea acestuia este reprezentat de un proiect individual dezvoltat de o singură persoană. Însă acest limbaj a evoluat de la un proiect simplu datorită contribuțiilor oferite de diverși programatori și de comunitatea care a fost creată în jurul acestui limbaj. Multe persoane care au contribuit la dezvoltarea acestui limbaj au hotărât să rămână anonimi.

2.1.1. Concepte generale ale limbajului Python

Python rulează pe diverse platforme (Linux, Windows, MacOS), fără a fi nevoie de modificări ale codului pentru a suporta aceste platforme în paralel. Limbajul este unul dinamic, adică tipurile de date sunt determinate în timpul execuției codului, oferind o structură flexibilă prin posibilitatea de a modifica tipul variabilelor pe parcursul rulării unei aplicații.

Un alt concept important este gestionara automată a memoriei, un sistem integrat în limbaj ceea ce oferă o dezvoltare rapidă a aplicațiilor fără a fi nevoie de a specifica în detaliu cum este gestionată memoria de aplicație.

2.1.2. Avantajele utilizării limbajului Python

Python oferă diverse avantaje programatorilor pentru o varietate mare de proiecte indiferent de experiența acestora.

- Productivitate: Fiind un limbaj interpretat[3] fără a fi necesară compilarea codului înainte de rulare. Python permite un proces rapid de *editare, debug* și *testare,* unde orice eroare de sintaxă sau limbaj este evidențiată imediat facilitând o corectare rapidă. Dacă performanța unei părți din aplicație este esențială, Python oferă posibilitatea optimizării codului prin integrarea de extensii scrise direct în limbajele **C** sau **C++,** eventual prin utilizarea a diverse **biblioteci** specializate pentru anumite implementări de cod.
- Biblioteci standard: Python dispune de un număr mare de biblioteci și instrumente care acoperă multe domenii cum ar fi manipularea de fișiere, networking, baze de date, inteligență artificială și operații matematice.
- Flexibilitate: Este un limbaj general care se pliază pe foarte multe domenii datorită instrumentelor de care dispune. Pentru că acest limbaj poate fi utilizat pentru a creea baze de date, servere, business logic și interfețe grafice în final se poate utiliza strict **Python** pentru a dezvolta o aplicație completă care folosește diverse arhitecturi și sisteme.
- Comunitate: Deține o comunitate oficială la nivel global, unde se pot găsi diverse materiale, tutoriale și documente utilizate de programatori începători dar și avansați pentru a dezvolta aplicații în diverse domenii. Fiind un proiect **open-source,** acest limbaj evoluează rapid prin contribuția comunităților și a organizațiilor.
- Sintaxă: Acest limbaj este ușor de citit, nu are restricții majore de scriere în comparație cu alte limbaje precum C++ și Java, unde este necesară utilizarea unor simboluri precum acoladelor, punct și virgulă la final de linie. Această simplicitate permite scrierea de cod mai eficientă, cantitatea de cod este redusă și mai ușor de înțeles ceea ce oferă o productivitate ridicată.

[3] Executare a codului linie cu linie fără compilare prealabilă

2.1.3. Dezavantajele utilizării limbajului Python

Având în vedere avantajele pe care acest limbaj le oferă există și dezavantaje și limitări care ar trebuii luate în vedere:

- **Memorie**: Python gestionează memoria automat prin **garbage collector,** dar această flexibilitate vine cu un cost. Acest limbaj folosește mai multă memorie față de alte limbaje de exemplu o variabilă de tip **Integer** în Python ocupă mai multă memorie față de una de tip **Int** în limbajul C++.
- **Viteză**: Unul dintre cele mai cunoscute dezavantaje ale acestui limbaj este viteza redusă de execuție. Acesta fiind mai lent față de orice limbaj (C++, Java), motivul principal pentru această situație este că Python execută fiecare instrucțiune, linie de cod la **runtime**[4] în loc să fie executate înainte de runtime de către un **compilator**
- **Dezvoltare de jocuri**: Nu este folosit pentru dezvoltarea de jocuri, chiar dispune de biblioteci care suportă dezvoltarea de jocuri, acest fap teste datorat dezavantajelor precizate anterior, lipsa performanței de executare a codului și de gestionare a memoriei, acest limbaj de programare nu este optim pentru dezvoltarea de jocuri.
- **Global Interpreter Lock**: Acesta este un dezavantaj tehnic, există un mecanism care permite executarea codului folosind un singur **thread**[5]**.** Mai ușor de înțeles, python nu permite rularea în paralel a codului ceea ce rezultă în dificultatea de a dezvolta aplicații care necesită o putere computațională mare bazată pe procesor. Însă acest lucru poate fi evitat prin folosirea modulului **multiprocessing** în loc de thread-uri, pentru a putea profita de toată puterea procesorului.

2.1.4. Baze de date de tip graf folosind Python

Integrarea bazelor de date de tip graf în **Python** este posibilă datorită bibliotecilor existente și a suportului oferit de diverse platforme.

Înainte să discutăm strict de integrarea bazelor de date de tip graf în **Python**, în continuare voi prezenta integrarea a două baze de de date pentru a se înțelege care este procesul general de **integrare** în **Python** a unei baze de date.

PostgreSQL este o bază de date **relațională** care stochează datele sub formă de tabele fiind potrivită pentru structuri de date bine definite, Integrarea în Python se face prin utilizarea bibliotecii **Psycopg2,** care oferă funcționalitățile necesare conectării și interacțiunii cu o bază de

[4] Momentul în care instrucțiunile sunt inițializate/executate.

[5] Fir de execuție sau secvență individuală de instrucțiuni pe care un procesor le execută.

date instanțiată pe platforma **postgres.** Următorul cod explică în detaliu cum cum se conectează la o bază de date folosind biblioteca **psycopg2.** Se completează credențialele pentru conectare a unei baze de date folosind funcția **connect** din biblioteca **psycopg2,** următorul pas este crearea unui **cursor** (funcție care execută instrucțiuni text în format SQL). Se utilizează funcția **execute** care primește ca parametru codul SQL care v-a fi executat de către baza de date. În final rezultatul acestei interogări este procesat de functia **fetchall()**

```
import psycopg2
def simple_query():
    conn = psycopg2.connect(
        dbname="user",
        user="postgres",
        password="1234",
        host="localhost"
    )
    cur = conn.cursor()
    cur.execute("SELECT user_id FROM Users")
    cur.fetchall()
    cur.close()
    conn.close()
```

Figure 5 Exemplu funcție folosind psycopg2

Oracle Database este un alt exemplu de bază de date relațională. Integrarea în Python s-a făcut prin utilizarea bibliotecii **oracledb** care permite conectarea la o bază de date de pe instanța **oracle** care permite executarea de interogări și manipularea datelor. Următorul exemplu este identic cu cel de de mai sus deoarece ambele biblioteci utilizează la bază aceeași logică.

```
import oracledb
def simple_query():
    conn = oracledb.connect(
        user="user",
        password="1234",
        dsn="localhost/XEPDB1"
    )
    cur = conn.cursor()
    cur.execute("SELECT user_ID FROM Users")
    cur.fetchall()
    cur.close()
    conn.close()
```

Figure 6 Exemplu interogare folosind oracledb în Python

Ultimul exemplu de integrare a unei baze de date este pentru una de tip graf oferit de platforma Neo4j. După cum se poate vedea în următorul exemplu, se folosește biblioteca **neo4j** și se importă clasa **GraphDatabase**. Se specifică datele de conectare cu funcția **driver,** se deschide o sesiune nouă, o interogare este executată și în final sesiunea este închisă. Acest proces este identic cu cele de mai sus.

```
from neo4j import GraphDatabase
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "1234"))
session = driver.session()
result = session.run("MATCH (u:User) RETURN u.user_id")
session.close()
driver.close()
```

Figure 7 Exemplu script de conectare la baza de date de tip graf Neo4j

Pe baza exemplelor de mai sus, integrarea unei baze de date de tip graf nu diferă față de una relațională, chiar dacă bibliotecile și denumirile funcțiilor sunt diferite, la bază au același proces: importare bibliotecă, inițializare conexiune, deschidere sesiune, executarea unei interogări, procesarea rezultatelor și în final închiderea sesiunii. Un programator care nu are experiență în integrarea unei baze de date de tip graf, în Python nu va avea dificultăți dacă acesta a integrat anterior alte tipuri de baze de date, procesul fiind unul asemănător.

În continuare voi prezenta un exemplu mai detaliat pentru integrarea unei baze de tip graf primul pas fiind crearea conexiunii la o instanță **Neo4j**. Se instalează biblioteca **neo4j** dacă nu este instalată prin comanda *pip install neo4j* comanda rulată în terminal sau cmd[6].

a) Crearea și stabilirea conexiunii: Se crează și instanțiază un obiect de tip **Driver** furnizat de biblioteca neo4j de exemplu (bolt://localhost:7687) și completarea datelor de autentificare *user* și *parolă.* Driverul este inițializat o singură dată și poate fi reutilizat ulterior pentru deschiderea și închiderea de sesiuni.
b) Verificarea conexiunii: Neo4j oferă o funcție de verificare a conexiunii, acest pas este opțional, funcția este accesată prin obiectul **Driver** creat anterior având denumirea **.verify_connection().**
c) Crearea unei sesiuni: : După ce conexiunea este stabilită, se deschide o nouă sesiune prin apelarea funcției **.session()** din obiectul Driver.
d) Interogarea bazei de date: Interogarea bazei de date se face prin funcția **.execute_query()**, această funcție primește un parametru de tip **String,** care reprezintă o instrucțiune sub forma limbajului **Cypher**. Interogarea poate fi de mai multe feluri:

[6] Command Prompt

creare, accesare și modificare a nodurilor, respective a relațiilor între acestea. Rezultatul unei interogări este mereu o *listă* sau *array* de obiecte. Un exemplu simplu este rezultatul acestei interogări:

```
MATCH (a:User)-[:FRIEND]->(b:User)
RETURN a.user_id AS from_user, b.user_id AS to_user
```

Rezultatul acestei interogări este o listă de obiecte, unde parametrii fiecărui obiect este accesat prin precizarea unui parametru. În cazul de față se iterează asupra listei și se accesează numele utilizatorului folosind parametrul "**from_user**" sau "**to_user**". Denumirile parametrilor sunt specificați în interogare la partea de **RETURN** fiind ușor de înțeles cum sunt citite și accesate datele în urma unei procesări a rezultatelor.

Aceștia sunt pașii de integrare a unei baze de date de tip graf, în cazul de față folosind platforma **Neo4j** în limbajul de programare Python.

2.2. Aplicație de tip rețea socială folosind Python și graph database.

O aplicație de tip rețea socială se bazează pe relații complexe între utilizatori și alte entități cum ar fi comunități, postări, comentarii, prieteni etc. În esență aceste conexiuni sunt un graf, indiferent dacă salvate într-o bază de date relațională sau una de tip graf. O bază de date relațională poate gestiona aceste conexiuni complexe între elemente însă o bază de date de tip graf este specializată în acest domeniu fiind o soluție optimă în acest context. Utilizarea limbajului Python oferă instrumente mature pentru dezvoltarea a diverse aplicații: PySide6, Tkinter, Kivy, BeeWareToga, WxPython (Fitzpatrick, 2025).

- Tkinter: Bibliotecă standard pentru interfețe grafice în Python folosit în special pentru aplicații simple având limitări majore din punct de vedere al flexibilității.
- wxPython: Oferă instrumente de bază pentru dezvoltarea unei aplicații simple și poate fi dezvoltată pentru orice tip de platformă (Linux, macOS și Windows).
- PySide6: Această bibliotecă oferă multe instrumente pentru dezvoltarea de aplicații complexe aceasta fiind utilizată în această lucrare deoarece este optimă pentru crearea de interfețe care conține layout-uri complexe.

Pentru a crea o rețea de tip socială în python se pot utiliza diverse biblioteci în funcție de obiectivele aplicației. Dacă este necesară o interfață complexă, bibliotecă PySide6 este ideală, aceasta fiind biblioteca folosită în dezvoltarea aplicației pentru lucrarea curentă. O altă bibliotecă

care poate fi utilizată în contextul unei rețele sociale, mai exact pentru o aplicație de tip mesagerie, biblioteca Tkinter este ideală deoarece nu este necesar crearea unei interfețe complexe.

Modelul general al unei aplicații sociale include o bază de date, un serviciu **back-end** în cazul de față limbajul Python și o interfață grafică unde utilizatorul poate interacționa cu aplicația. Această aplicație poate fi locală[7] sau accesibilă în browser. Baza de date poate fi relațională sau de tip graf unde procesul de integrare este unul simplu explicat anterior. Indiferent de tipul aplicației, alegerea bibliotecilor și a bazei de date este importantă și să respecte obiectivele aplicației. Pot fi luate în considerare diverse criterii pentru a stabili ce instrumente pot fi folosite precum:

- Complexitatea interfeței: Interfața grafică dorită poate influența ce bibliotecă este utilizată.
- Baza de date: Tipul bazei de date diferă de la o aplicație la alta, pentru o bază de date care nu folosește relații complexe, una de tip tradițională este ideală.
- Alte instrumente: În funcție de natura aplicației, diverse biblioteci sunt utilizate. Pentru creare de chart-uri cea mai populară bibliotecă este **Matplotlib**, care poate fi utilizată pentru reprezenta vizual performanța unei baze de date sau a unor funcții. **Numpy** o altă bibliotecă folosită des pentru procesarea și manipularea de date.

2.3. Avantajele și limitările integrării unei baze de date de tip graf pentru o rețea socială.

Dezvoltarea unei aplicații sociale este un proces complex unde există diverse limitări atât din punct devere a design-ului cât și a tehnologiilor utilizate.Avantajele utilizării unei baze de tip graf în contextul unei aplicații sociale sunt următoarele:

- Modelarea datelor: Rețele sociale sunt în esență grafuri, indiferent de baza de date utilizată ceea ce oferă un avantaj bazelor de date de tip graf, unde datele sunt create foarte ușor fără a fi nevoie de a modifica interacțiunea și relațiile acestora, fiind o soluție ideală în acest context.

- Eficiența interogărilor: Interogările cel mai des întâlnite în contextul unei aplicații sociale pot fi următoarele: *Care sunt prietenii prietenilor mei , Care sunt particularitățile comune cu utilizatorul X.* Datorită bazelor de date de tip graf, aceste interogări sunt foarte performante unde o bază de date relațională are dificultăți deoarece trebuie sa execute recursive interogări pentru a ajunge la același rezultat.

[7] Aplicație desktop pe un sistem de operare

- Scalabilitate: În anul 2021, Neo4j a demonstrat folosind tehnologia super-scaling pentru a demonstra performanța acestei platforme pe o bază de date cu peste 200 miliarde de noduri și mai mult de un trilion de relații. Acest număr noduri și date poate cuprinde toți utilizatori care au acces la o aplicație de tip rețea socială, inclusiv relațiile complexe între aceștia (Neo4j, Neo4j Breaks Scale Barrier with Trillion+ Relationship Graph, 2021).

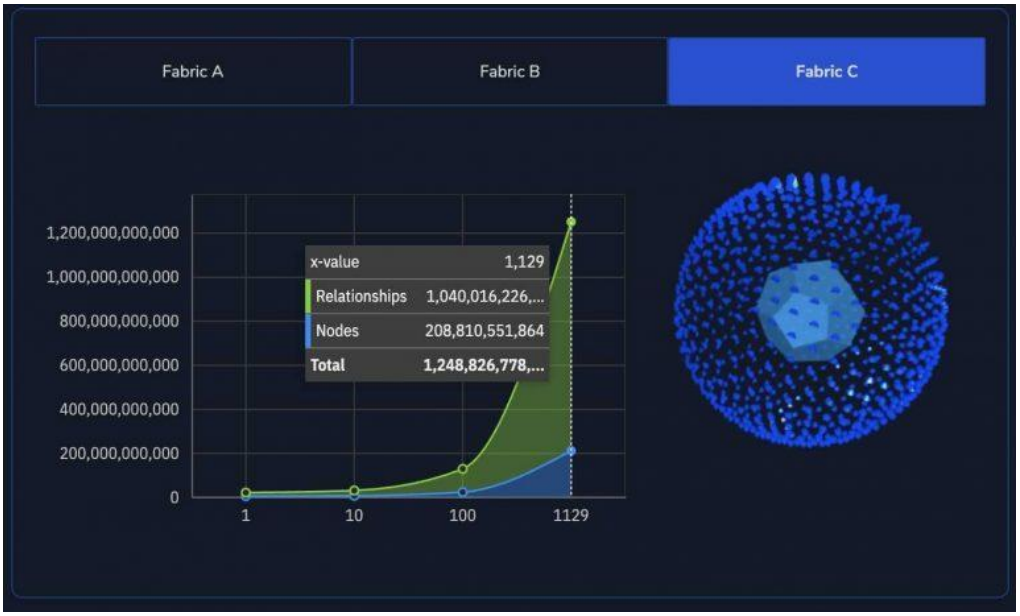


Figure 8 Neo4j breaks scale barrier with trillion+ relationship graph using distributed architecture, advanced sharding (https://neo4j.com/press-releases/neo4j-scales-trillion-plus-relationship-graph)

- Interogări declarative: Neo4j folosind limbajul Cypher, interogările pot fi expresive ceea ce au un avantaj în manipularea unui model de tipă rețea socială.

Având în vedere numărul mare de avantaje pe care bazele de date de tip graf le oferă, există și dezavantaje sau limitări:

- Scalabilitate orizontală: Una dintre cele mai comune limitări a unei baze de tip graf este scalarea pe orizontală adică utilizarea bazei de date pe mai multe dispozitive. În prezent există mai multe soluții pentru a evita această limitare însă tot nu se pot compara cu bazele de date relaționale.
- Particularități: Bazele de date de tip graf utilizează limbaje specifice platformei din care fac parte, având un impact major pentru un programator care decide să învețe noi tehnologii legate de baze de date de tip graf.
- Fișiere mari: Bazele de date nu sunt ideale pentru stocarea de fișiere mari sau a conținutului de mărimi mari de exemplu videoclipuri. În Neo4j, se stochează doar

adresele de unde să fie accesate aceste fișiere și se utilizează un alt mediu de stocare pentru acestea.

Utilizarea unei baze de date de tip graf pentru dezvoltarea unei aplicații sociale are multe avantaje în ceea ce privește structura modelului de date și eficiența interogărilor transversale.

2.4. Înterogări complexe în baze de date de tip graf

Datorită limbajului Cypher și bazei de date de tip graf se pot crea interogări complexe pentru o rețea de tip socială, unde cel mai popular tip de interogare este multi-hop. „*Pentru o bază de date relațională acest tip de interogare constă în utilizarea a multiple joncțiuni între tabele ceea ce duce la blocaje de performanță pe măsură ce dimensiunea datelor și a relațiilor cresc. Interogările de tip multi-hop solicită de obicei traversarea a mai multor noduri printr-o singură interogare. De exemplu un lanț de aprovizionare, o interogare multi-hop ar putea urmări modul în care un produs este transportat prin diverşi furnizori* ” (Palifka, 2024).

Un exemplu de interogare **multi-hop** în contextul unei baze de date de tip graf pentru o rețea socială este:

```
MATCH (p:Post)<-[:CREATED]-(u:User)
OPTIONAL MATCH (p)-[:HAS_POST]->(c:Community)
RETURN p.id AS id,
    p.title AS title,
    p.content AS content,
    p.timestamp AS timestamp,
    p.visibility AS visibility,
    p.likes AS likes,
    p.dislikes AS dislikes,
    p.comments AS comments,
    u.username AS author_name,
    u.id AS author_id,
    c.name AS community_name
ORDER BY p.timestamp DESC
```

În această interogare, există două **hop-uri,** primul hop fiind exact prima linie de cod din exemplu și al doilea hop a doua linie. Termenul **opțional** implică faptul că nu este obligatoriu ca relația respectivă să existe iar în final se returnează datele necesare cu parametri specificați.

O altă interogare complexă este **pattern-matched,** această interogare caută un anumit tip model care conține un set de relații.

```
MATCH (u1:User)-[:MEMBER_OF]->(c:Community)<-[:MEMBER_OF]-(u2:User)
WHERE u1.id <> u2.id
RETURN u1.username AS user1, u2.username AS user2, c.name AS common_community
```

Scopul acestei interogări este de a căuta doi utilizatori care fac parte din aceeași comunitate. Acest tip de interogare este util pentru a recomanda prieteni pentru anumiți utilizatori car fac parte din aceeași comunitate, un caz real sunt notificările de pe platforma Facebook care conține mesajul *Poate cunoști această persoană* unde se folosește acest tip de interogare (în cazul unei baze de date de tip graf) pentru a găsi persoane care au trăsături asemănătoare cu utilizatorul țintă.

CAPITOLUL 3. Dezvoltarea aplicației de tip rețea socială TOPIKKA

Acest capitol constă în prezentarea aplicației TOPIKKA, o aplicație de tip rețea socială care utilizează python pentru straturile de interfața utilizator si logica aplicației, respectiv o bază de date de tip graf. Aplicația se bazează pe formarea de conexiuni între utilizatori și postările din comunități prin utilizarea unui algoritm de recomandare.

Din perspectiva tehnologică, soluția curentă, constă în utilizarea unei interfețe grafice de tip desktop realizată în Python folosind biblioteca **PySide6** împreună cu o bază de date de tip graf oferită de platforma **Neo4j.** Motivarea alegerii unei baze de tip graf este susținută de natura aplicației unde se pot crea relații complexe între entități fiind o situație ideală pentru utilizarea acestei tip de baze de date. Neo4j permite stocarea entităților sub formă de noduri și a legăturilor sau relațiilor între aceste entități sub formă de *muchii* sau *arce.*

Interfața grafică dezvoltată cu **PySide6** oferă utilizatorilor o platformă complexă din punct de vedere al interacțiunii, însă se pot face argumente legat de natura interfeței grafice, de exemplu utilizarea unei *pagini web* față de o aplicație de sine stătătoare pe un sistem de operare (Windows, Linux sau MacOS). Un argument pentru utilizarea unei aplicații de tip desktop și nu o pagină web este: Biblioteca **PySide6** utilizează fișiere cu extensia **.ui** (user interface), unde aceste fișiere conțin cod **XML** (Extensible Markup Language). Acest tip de fișiere pot fi convertite destul de ușor în format **HTML.** Dacă este necesară convertirea platformei de la o aplicație de sine stătătoare pe un sistem de operare la o pagină web, pașii necesari pentru a îndeplini acest proces sunt puțini și simpli.

Aplicația prezintă un **prototip de rețea socială** de tip desktop, folosind o arhitectura simplă, internă unde se folosește o interfață locală, un set de logici de procesare a datelor și baza de date de tip graf. Nu există un server web extern. Logica de **backend** este încorporată în aplicație folosind scripturi **python** care comunică cu baza de date **Neo4j**.

Proiectarea acestui sistem a pornit de la definirea clară a obiectivelor și cerințelor de dezvoltare, urmată de proiectarea interaței folosind un instrument extern. Astfel s-a asigurat un proces de dezvoltare simplu unde fiecare funcționalitate majoră fiind anticipată datorită cerințelor stabilite de design-ul aplicației și obiectivele clar definite de la început.

3.1. Cerințele funcționale ale aplicației

Obiectivul acestei aplicații este de a crea un prototip de tip rețea socială folosind ca arhitectură de bază limbajul de programare **Python** împreună cu o bază de date de tip graf, în cazul de față este utilizată platforma **Neo4j.** Pentru ca utilizatorul să interacționeze cu o interfață grafică a fost utilizată biblioteca **PySide6.** Toate aceste instrumente și platforme sunt utilizate

pentru a livra în final o aplicație complexă, complet funcțională cu minimul de funcționalități care oferă o experiență asemănătoare cu alte aplicații de tip rețea socială de pe piață.

Funcționalitățile principale ale aplicației au fost identificate în planul de proiectare a aplicației și clasificate pe niveluri de prioritate având ca scop dezvoltarea aplicației ca MVP (Minimum Viable Product)

În faza de planificare, obiectivele și funcționalitățile au fost identificare și clasificate pe niveluri de prioritate având ca scop dezvoltarea aplicației ca **MVP** (Minimum Viable Product) .

Funcționalitățile stabilite sunt următoarele:

- Crearea unui profil de utilizator: Permite utilizatorilor să creeze conturi
- Postări și comentarii: Utilizatorii pot crea comentarii la postări.
- Comunități: Postările pot fi create doar în cadrul unei comunități.
- Butoane interactive: Implementarea logicii de **like** pentru postări și comentarii
- Algoritm de recomandare: În funcție de interacțiunile utilizatorului cu postări sau comentarii sunt recomandate postări.

Prin implementarea acestor functionalitati folosind limbajul de programare **Python** împreună cu platforma **Neo4j** care oferă o bază de date de tip graf obiectivele stabilite anterior sunt atinse și aplicația poate fi livrată ca **MVP**.

3.2. Tehnologiile alese pentru implementarea aplicației

Pentru dezvoltarea unei aplicații de tip rețea socială, a fost necesară selectarea și integrarea a unor tehnologii și instrumente care să îndeplinească obiectivele stabilite anterior. Alegerea acestor instrumente au fost influențate de obiectivele aplicației și funcționalitățile necesare pentru a livra un prototip MVP. Instrumentele folosite sunt următoarele:

a) **Python**: limbajul de programare utilizat pentru această aplicație.
b) **PySide6**: Bibliotecă oferită de Python pentru dezvoltarea de interfețe grafice. Motivul pentru utilizarea aceste biblioteci este complexitatea design-ului care poate fi creat fără restricții majore.
c) **Neo4j**: Baza de date de tip graf utilizată pentru gestionarea entităților și a relațiilor, fiind un sistem matur și optimizat, este o alegere bună pentru acest tip de aplicație.
d) **Figma**: Platformă utilizată pentru proiectarea de interfețe grafice în procesul de proiectare a unei aplicații. Permite crearea unei interfețe complexe și instrumente de export, acestea fiind utile în aplicațiile de tip web dar și pentru stabilirea unor interfețe complexe care poate fi replicat de biblioteca PySide6 ulterior.

Prin utilizarea acestor tehnologii aplicația este dezvoltată modular unde fiecare tehnologie a contribuit la acest proces fiind posibil dezvoltarea aplicației în continuare chiar și după ce obiectivele de MVP au fost îndeplinite. Convertirea acestei aplicații de la una locală de tip desktop la o pagină web se poate face ușor și rapid folosind instrumente de convertire a design-ului creat în **Figma** sau a celui replicat în biblioteca **PySide6**. Logica codului între interacțiunea aplicației cu baza de date poate fi refolosit indiferent de platforma utilizată, în cazul unei pagini web, pentru a accesa arhitectura de **backend** se pot utiliza funcții API pentru accesarea datelor necesare.

3.3. Proiectarea interfeței utilizator.

Pentru proiectarea unei interfețe grafice unde utilizatorul interacționează cu aplicația sunt utilizate mai multe instrumente cum ar fi **Figma,** un instrument extern pentru stabilirea unui model de bază al interfeței care este utilizat și dezvoltat ulterior de biblioteca **PySide6** pentru a creea interfețele/paginile aplicației.

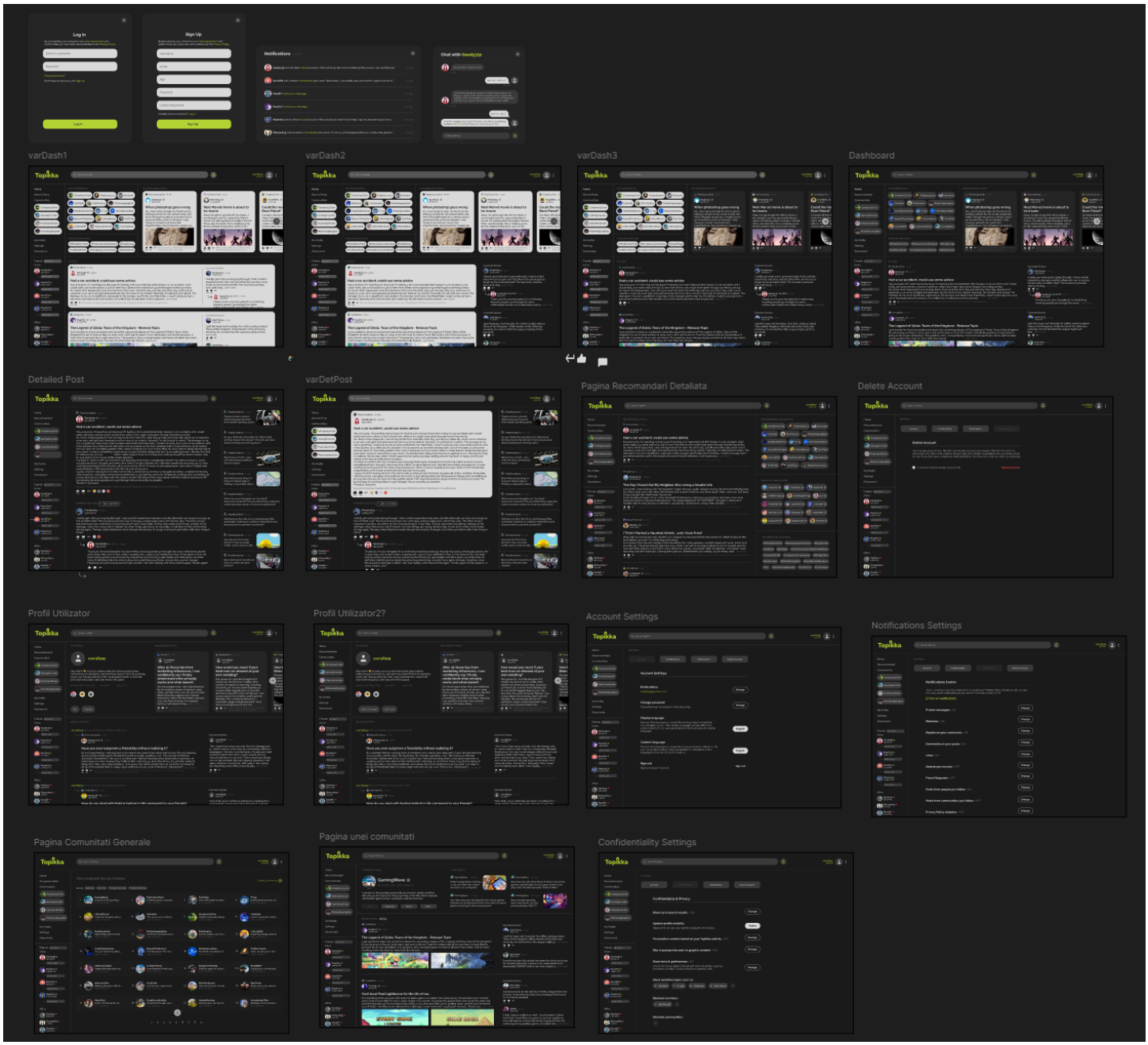
În procesul de proiectare a interfețelor utilizator au fost luate în calcul funcționalitățile de bază stabilite la proiectarea aplicației de exemplu: postările au obligatoriu o comunitate ca referință. Având în vedere funcționalitățile respective, proiectarea unui model de interfață, v-a respecta din start obiectivele și scopul stabilit anterior.

3.3.1. Crearea unui prototip al design-ului folosind instrumente externe.

Figma este o platformă pentru crearea de prototipuri din punct de vedere al design-ului. Folosind acest tool, se poate *crea, testa și distribui* interfața unei aplicații de tip pagină web, aplicație pe un sistem de operare sau chiar aplicații pe platforme Android. Oferă diverse instrumente de proiectare precum animații, scripturi etc. Figma oferă un mediu de dezvoltare a unei interfețe complexe folosind diverse *elemente și containere*[8].

[8] https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma

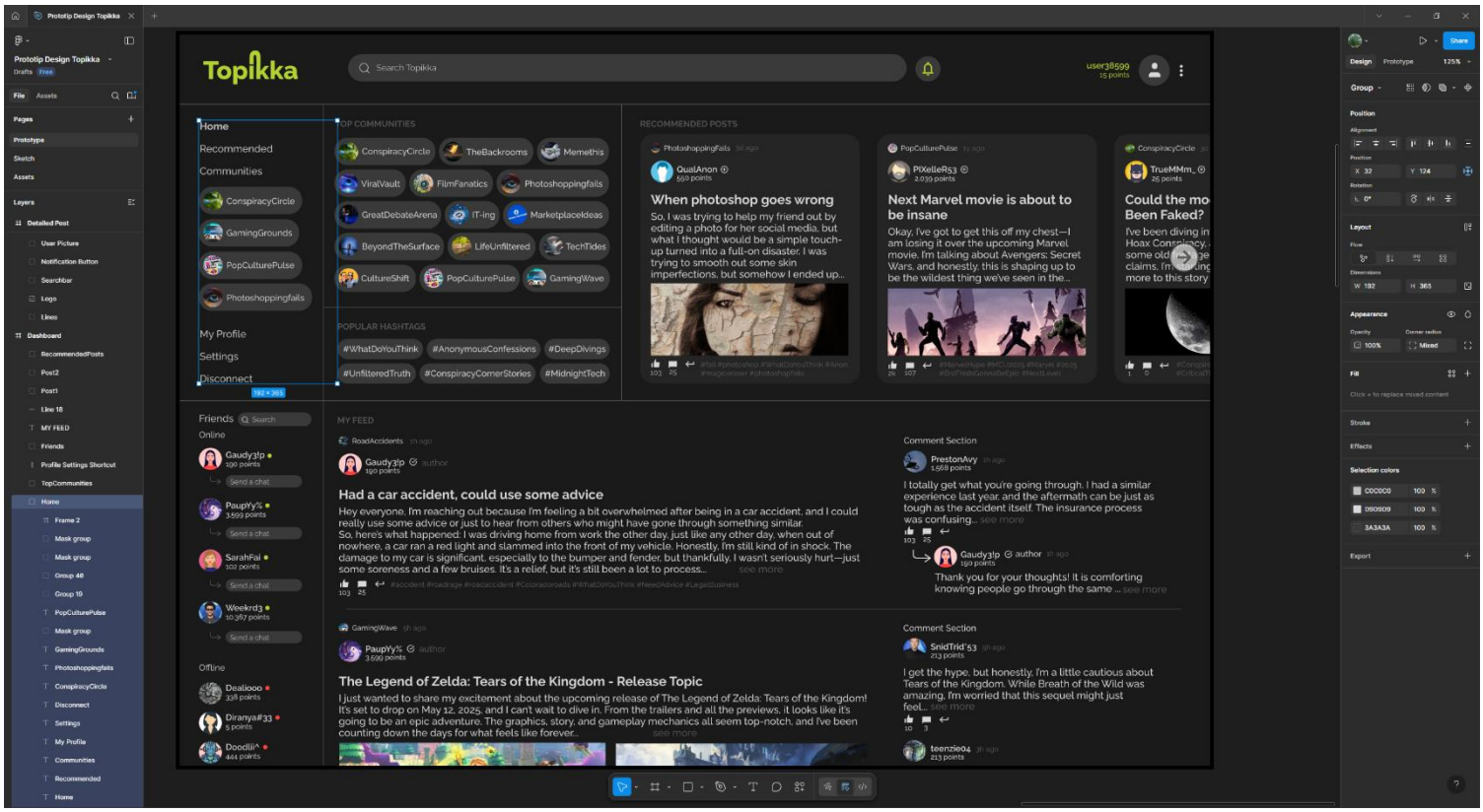
Prin folosirea acestui instrument au fost create diverse prototipuri pentru ferestrele (paginile) aplicației. Deoarece design-ul acestei aplicații este unul original, paginile conțin layout-uri[9] complexe.



Figură 9 Prototipul interfetei utilizator în Figma

După cum se poate vedea în *Figura 9 Prototipul interfeței utilizator în Figma* au fost create diverse prototip-uri pentru diferite pagini: logare, creare cont, pagina principală, pagina de vizualizare a postării în detaliu. Pentru pagina principală și pagina de vizualizare a unei postări sau creat pagini utilizând diverse culori principale, acest lucru a ajutat la alegerea unei teme generale fiind una închisă la culoare fără a exista un contrast major între culori.

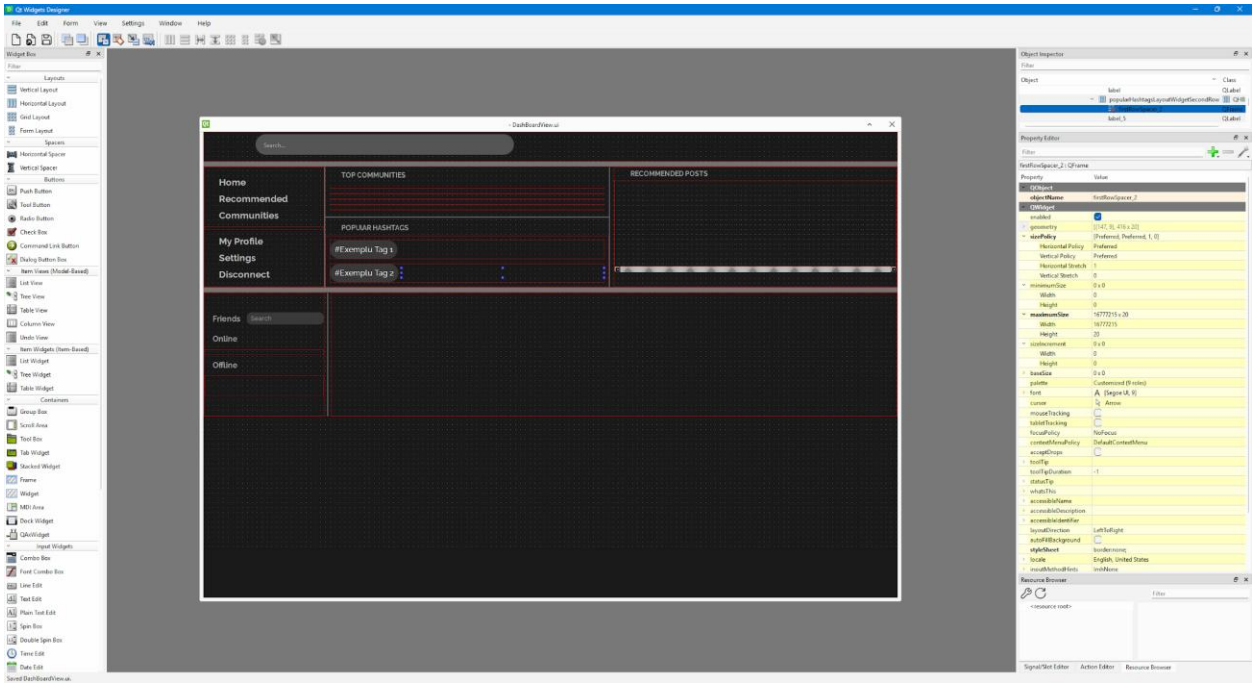
[9] Modul în care sunt aranjate din punct de vedere vizual elementele în pagină.



Figură 10 Prototip design detaliat a unei pagini

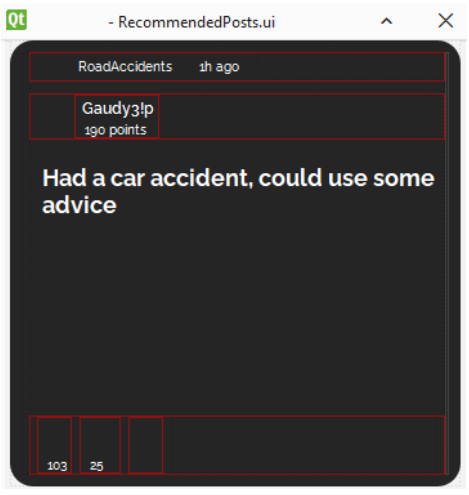
În figura de mai sus, interfața a fost creată folosind diverse elemente oferite de figma, de la elemente de tip text, butoane la imagini și forme complexe. Datorită utilizării platformei **Figma** proiectarea unei interfețe a fost unul ușor și care menține funcționalitățile necesare de livrare a acestei aplicații.

În continuare, interfața dezvoltată în Figma a fost recreată în **Designer,** instrument oferit de biblioteca **PySide6,** instrument asemănător cu Figma dar care are limitări majore legat prezentarea unui design. Acest tool ajută la crearea unei structuri de bază utilizată ulterior de clase în Python care aduce la viață o pagină prin adăugarea de elemente personalizate.



Figură 11 Exemplu prototip interfață în Designer

În figura de mai sus, a fost creat structura de bază a paginii și a layout-urilor (containere care conțin **widget-uri** sau forme complexe). Acest prototip este salvat ulterior într-un fișier de tip **.ui** și convertit folosind biblioteca PySide6 într-o clasă Python care este utilizată ulterior pentru afișarea unei pagini în aplicație. Folosind acest tool se pot crea și widget-uri specifice, exemplu în figura următoare:



Figură 12 Exemplu widget complex în Designer

Acest **widget** sau element complex este utilizat pentru a afișa multiple postări recomandate, scopul creării acestui element este de a fi reutilizat într-o funcție de tip **loop** pentru a popula o anumită zonă din interfață cu elemente de tip **postări recomandate.** Folosind această strategie se

pot crea diverse widget-uri specifice care sunt reutilizate de mai multe ori în interfață pentru a minimiza cantitatea și calitatea codului scris.

3.4. Implementarea aplicației

Implementarea aplicației constă în mai mulți pași care trebuie urmații. În continuare voi discuta în detaliu cum a fost implementă aplicația pas cu pas pentru a livra un prototip funcțional care conține funcționalitățile necesare.

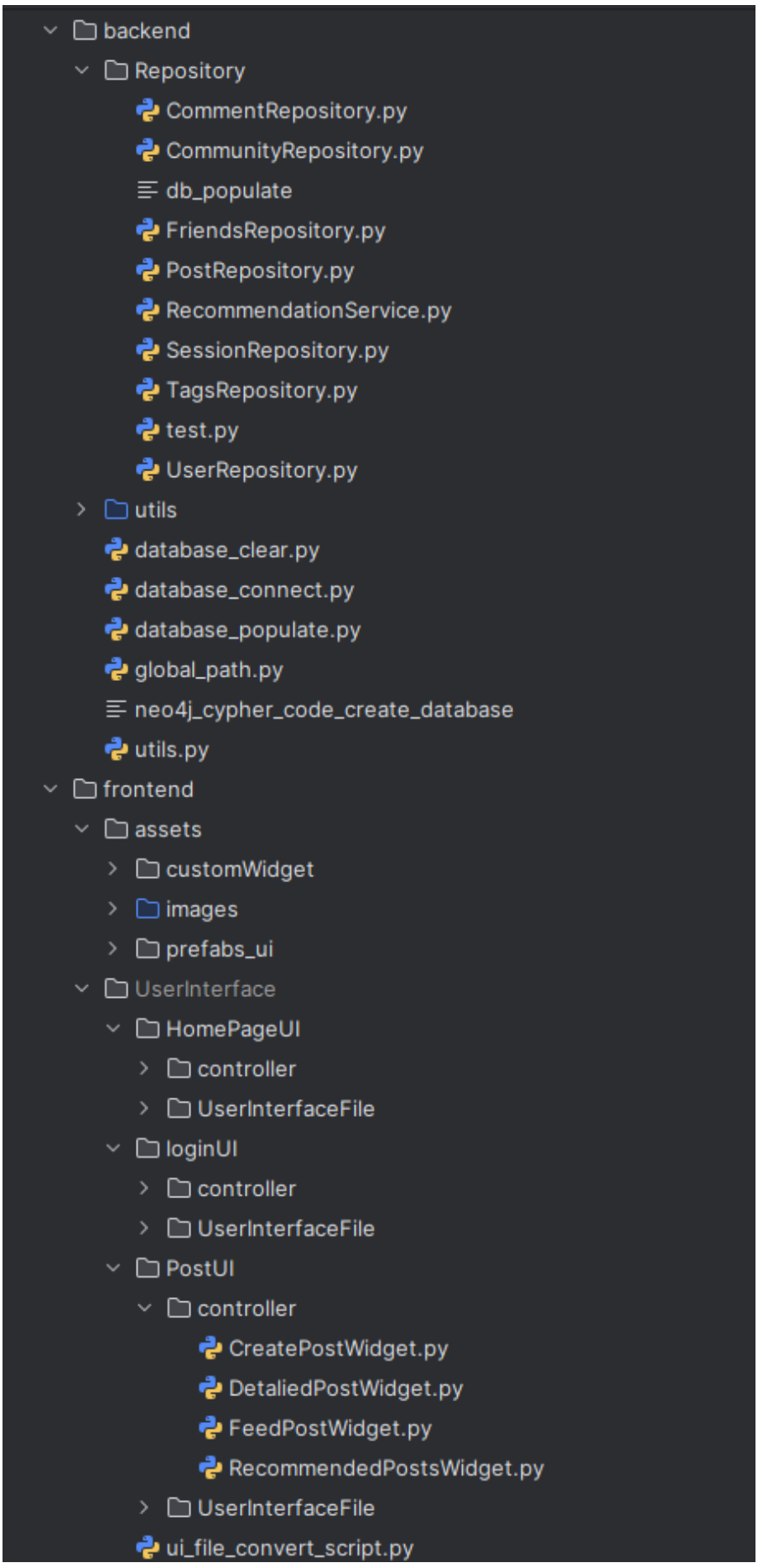
3.4.1. Structura aplicației

Proiectul este organizat pe mai multe module care reflectă structura acestei aplicații**.** Modulul **frontend** conține logica ce ține de interfața cu care utilizatorul interacționează iar modulul de **backend** gestionează interacțiunea cu baza de date și interfață. Fiecare modul conține la rândul său alte foldere pentru a păstra o coerență în structura aplicației.

După cum se poate vedea în figura de mai jos *Figură 4 .Structura completă a aplicației TOPIKKA*, în folderul **backend > repository,** sunt regăsite fișiere care gestionează fiecare entitate din baza de date, un exemplu **UserRepository** conține funcții de *read, create, delete și update ,* funcții de bază pentru gestionarea unei entități. Există de altfel și funcții complexe, personalizate în funcție de funcționalitățile implementate în aplicație, de exemplu **RecommendedService** care conține o funcție care returnează *postări* recomandate.

Interfața aplicației se află în modulul frontend care contine folderul **assets** și alte foldere pentru fiecare tip de fereastră (pagină de logare, creare cont, pagină principală și pagină de vizualizare a unei postări în detaliu). Fiecare folder care reprezintă o fereastră (pagină) conține la rândul său două secțiuni **controller** și **UserInterface**, unde în **controller** se află logica de interacționare a interfeței iar **UserInterface** conține fișiere statice de design create din fișiere .ui (user interface) dezvoltate folosind instrumentul **Designer** [10]oferit de biblioteca **PySide6**.

[10] Aplicație interactivă de crearea interfețe folosind funcții de drag-and-drop.



Figură 13 .Structura completă a aplicației TOPIKKA

3.4.2. Modelarea și implementarea bazei de date de tip graf

După proiectarea interfeței utilizator, următorul pas a fost crearea bazei de tip graf. În principiu s-a urmărit crearea nodurilor și a relațiilor care să respecte funcționalitățile stabilite la proiectarea aplicației și care să respecte interfața utilizator de bază dezvoltată în platforma **Figma**. Fiind o aplicație de tip rețea socială, nodurile principale sunt:

- **User**: reprezintă utilizatorii cu parametrii (username, email, points, status, reputation, img, și password.
- **Comment**: comentarii cu parametrii (text, likes, dislikes, timestamp)
- **Session**: interacțiunea utilizatorului cu aplicația cu parametrii (id, last_active și status)
- **Community**: reprezintă comunitățile cu parametrii (name, icon, description, points)
- **Post**: postări care sunt create în cadrul unei comunități cu parametrii (title, content, likes, dislikes, timestamp și visibility)
- **Tag**: reprezintă tag-urile care au doar un parametru (name).

Relațiile au fost stabilite ulterior între noduri:

- **friend_with:** relație între nodurile utilizatori care reprezintă prietenia dintre aceștia.
- **member_of:** relație între nodul **User** și **Community** care reprezintă că userul face parte din comunitatea respectivă.
- **created:** relație între **User** și nodurile **Post/Comment** implică crearea unei postări sau comentariu de către un utilizator.
- **Liked:** interacțiunea între utilizator și reputația unui comentariu sau postări.
- **has_session:** reprezintă interacțiunea utilizatorului cu aplicația.
- **interacted_with:** reprezintă interacțiunea unui utilizator cu o postare sau comentariu care v-a afecta algoritmul de recomandare.
- **has_post:** Legătura dintre comunități și postări care implică faptul că postarea a fost creată în comunitatea respectivă.
- **Tagged_as:** Legătura dintre tag-uri și nodurile **User**, **Community** și **Post**.
- **has_comment:** Reprezintă legătura dintre o postare și comentariile pe care le deține.
- **Reply_to:** Legătura recursivă între comentarii. Un comentariu poate fi un răspuns la un alt comentariu și nu doar la o postare.

După stabilirea nodurilor și a relațiilor între acestea, a fost creată baza de date de tip graf cu date pentru a verifica interacțiunea nodurilor și logica bazei de date. Un exemplu ușor de vizualizat ar fi crearea unui utilizator folosind următorul script Cypher:

```
MERGE (user38599:User {
    id: '38599',
    username: 'user38599',
    points: 15, email: 'user38599@topikka.com',
    img: 'avatars/default_av.png',
    status: 'online',
    password: 'hashed_password'});
```

Acest script crează un utilizator folosind parametrii specifici, acest script este utilizat ulterior de clasa **UserRepository** cu scop de a crea utilizatori. În continuare voi prezenta un alt script pentru crearea unei relații între nodul **utilizator** și o **comunitate:**

```
MATCH (u:User {username: 'Gaudy3!p'}),
      (c:Community {name: 'RoadAccidents'})
MERGE (u)-[:MEMBER_OF]->(c);
```

În scriptul de mai sus a fost creată o relație *member_of* între utilizator și o comunitate. Procesul acesta este repetat pentru toate nodurile și relațiile existente. În final după ce baza de date este populată, arată în felul următor:

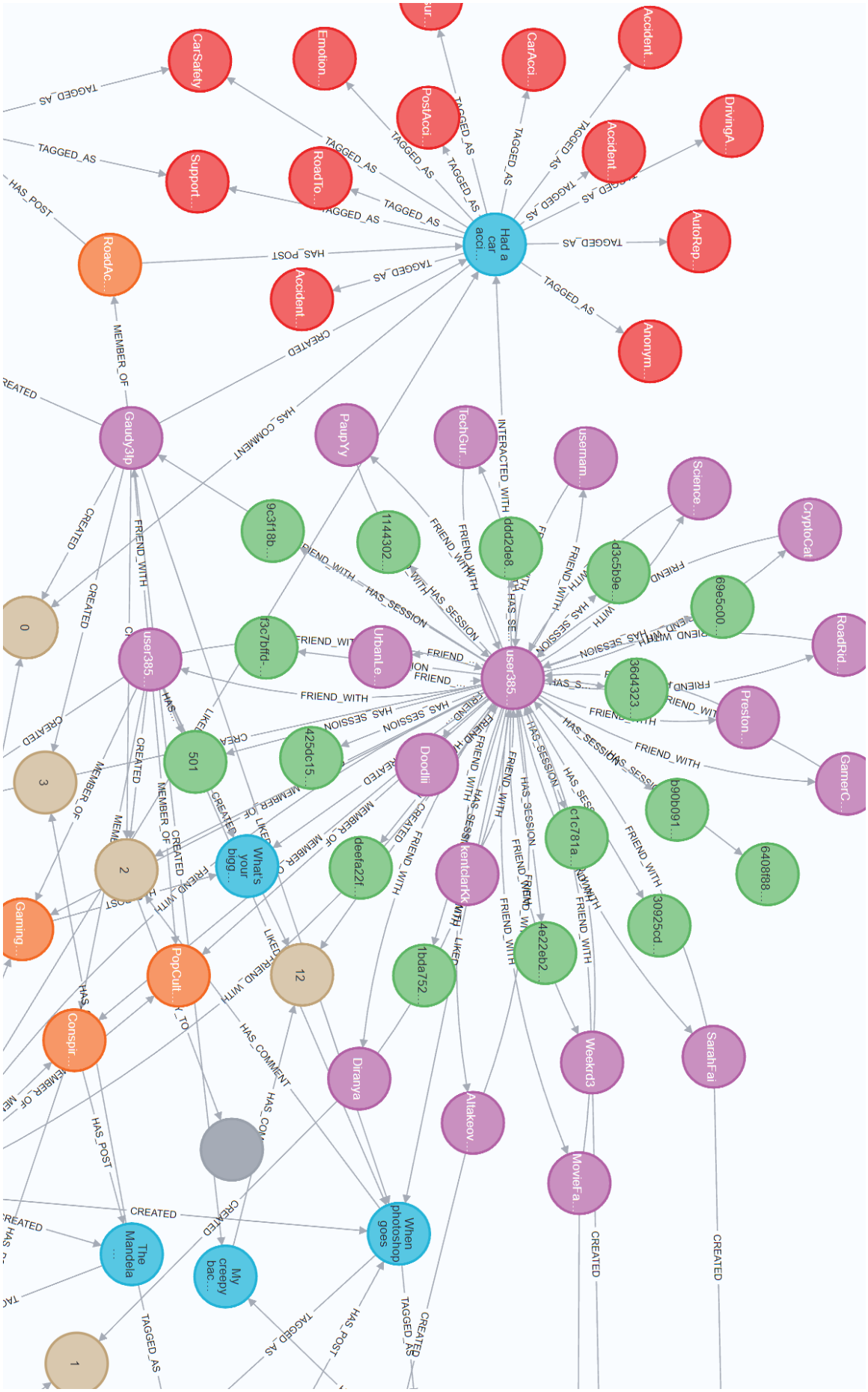


Figura 14 Reprezentare vizuală a bazei de date de tip graf

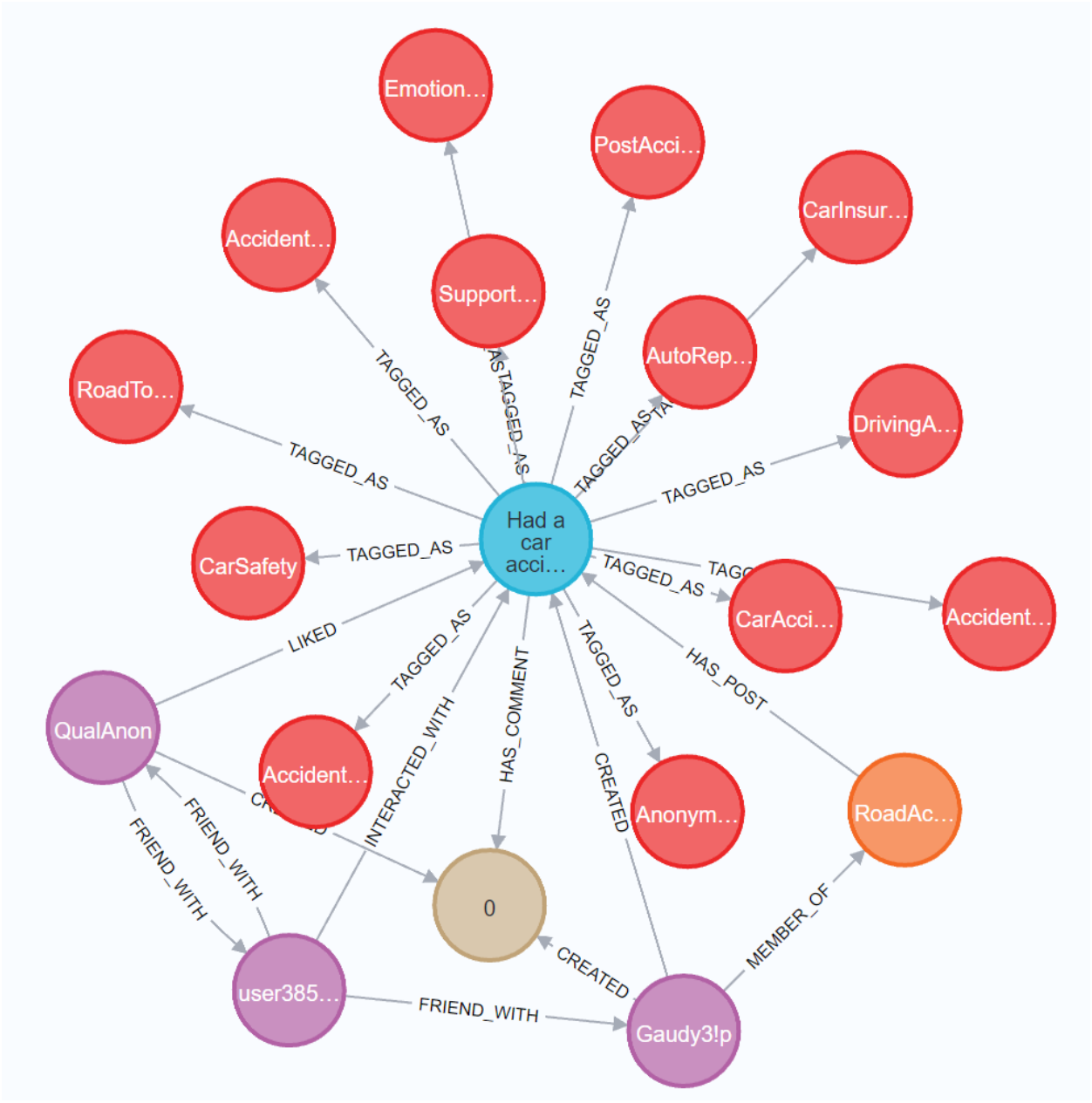


Figura 15. Reprezentare vizuală a unei postări și legăturile cu alte noduri

3.4.3. Dezvoltarea modulului repository

Acest pas constă în crearea a diferitor clase în **Python** pentru accesarea bazei de date de tip graf și utilizarea a diverse funcții de tip **CRUD** (create, read, update, delete), funcții de bază pentru interacțiunea cu baza de date.

Au fost create diverse clase repositoy pentru (user, community, post, comment, session și tag), iar în final au fost create alte clase repository personalizate precum (recommandation și friends). Un exemplu de clasa repository este următorul:

```python
def read_user(driver, email):  1 usage  Cojocariu Daniel
    """
    Retrieve a user node based on the email.
    """
    with driver.session() as session:
        with session.begin_transaction() as tx:
            read_query = """
            MATCH (u:User {email: $email})
            RETURN u.id AS id, u.email AS email, u.password AS password, u.img AS img, u.status AS status
            """
            result = tx.run(read_query, email=email)
            user = result.single()

            if user:
                return {
                    "id": user["id"],
                    "email": user["email"],
                    "password": user["password"],
                    "img": user["img"],
                    "status": user["status"]
                }
            return None
```

Figură 16 Exemplu funcție în UserRepository

Această funcție are ca rol returnarea unui **nod utilizator** în funcție de **email-ul** acestuia. Variabila read_query este un **string** care reprezintă explicit un cod în limbajul Cypher unde rezultatul este un obiect/element care conține datele utilizatorului. Această funcție este utilizată pentru a accesa datele unui utilizator.

Pentru a folosi aceste funcții, *driver-ul neo4j* care se ocupă cu conectarea la baza de date este inițializat la pornirea aplicației și după fiecare clasă repository va utiliza acest driver pentru a deschide o nouă sesiune, executarea unei interogări, procesarea datelor și în final închiderea aceste sesiuni.

```
def get_recommended_posts(self, user_id, limit=10):  2 usages  Cojocariu Daniel *
    logging.critical("RUN Recommendation settings")
    with self.driver.session() as session:
        query = """
            MATCH (u:User {id: $user_id})-[i:INTERACTED_WITH]->(p:Post)
            WITH u, COLLECT(p.id) AS seen_post_ids
            MATCH (rec:Post)
            WHERE NOT rec.id IN seen_post_ids
            OPTIONAL MATCH (u)-[int:INTERACTED_WITH]->(:Post)-[:TAGGED_AS]->(t)<-[:TAGGED_AS]-(rec)
            WITH rec,
                 COALESCE(SUM(int.score), 0) AS score,
                 rec.likes AS likes,
                 rec.dislikes AS dislikes,
                 rec.comments AS comments
            RETURN rec.id AS id,
                   rec.title AS title,
                   rec.content AS content,
                   rec.timestamp AS timestamp,
                   COALESCE(likes, 0) AS likes,
                   COALESCE(dislikes, 0) AS dislikes,
                   COALESCE(comments, 0) AS comments,
                   score
            ORDER BY score DESC, timestamp DESC
            LIMIT $limit
        """
        result = session.run(query, user_id=user_id, limit=limit)
```

Figură 17 Exemplu funcție complexă din Repository

Această funcție este una complexă utilizată pentru a recomanda postări bazate pe interacțiunea utilizatorului cu alte postări sau comentarii. Postările recomandate sunt postări cu care utilizatorul nu a interacționat încă și sunt sortate în funcție de scorul de relevanță (calculat în funcție de tag-uri comune accesate de postările anterioare) și timpul publicării postării (cel mai recente). Pentru a explica în detaliu logica codului de mai sus voi explica fiecare proces important din funcție:

a) Se caută toate postările cu care utilizatorul a interacționat în trecut
b) Se salvează toate postările cu care utilizatorul a interacționat deja într-o listă (**seen_post_ids**).
c) Ulterior se caută toate postările cu care utilizatorul nu a interacționat încă.
d) Se caută postările candidate pentru recomandare au tag-uri asociate în comun cu postările anterioare accesate de către utilizator.
e) Se calculează un scor pentru fiecare postare recomandată. Dacă există legături prin tag-uri comune se calculează scorul relațiilor **interacted_with**. Dacă nu există legături scorul este 0.
f) Se ordonează de două ori, prima data după scor și a doua oară după data la care au fost create postările.

g) În final se returnează informațiile relevante postărilor, parametrii care sunt utilizați ulterior în interfață.

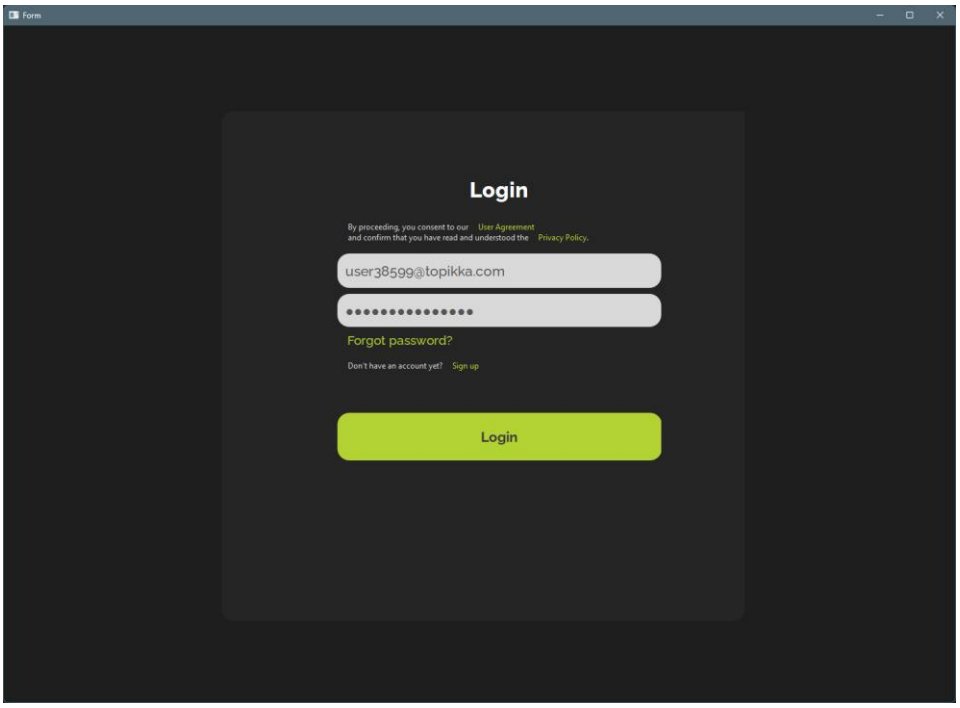
3.4.4 Implementarea interfeței grafice.

Dezvoltarea interfețelor din aplicație continuă după crearea modulului **Repository** prin implementarea claselor de tip **controller** în modulul **frontend**. Fiecare clasă se ocupă cu o interfață și logica din aceasta.

```
class LoginWindow(QWidget):  2 usages  Cojocariu Daniel
    def __init__(self):  Cojocariu Daniel
        super().__init__()
        self.ui = Ui_Form()
        self.ui.setupUi(self)
        self.new_window = None
        self.ui.loginButton.clicked.connect(self.start_login)

        # Use the singleton instance of the Neo4j driver
        self.driver = Neo4jDriverSingleton.get_driver()
        self.user_repo = UserRepository(self.driver)
        self.session_repo = SessionRepository(self.driver)
        self.session_id = None  # Store active session ID

        # Add the loading bar
        self.ui.usernameInput.setText("user38599@topikka.com")
        self.ui.passwordInput.setText("hashed_password")
        self.show()

    def start_login(self):  1 usage  Cojocariu Daniel
        """Start the login process with a loading animation."""

        # Get the username and password
        username = self.ui.usernameInput.text().strip()

        password = self.ui.passwordInput.text().strip()

        # Start the query in a separate thread
        self.query_thread = QueryThread(username, password, self.user_repo)
        self.query_thread.finished.connect(self.process_login)  # Connect the result to the handler
        self.query_thread.start()  # Start the query thread

    def process_login(self, user):  1 usage  Cojocariu Daniel
        """Perform the actual login process after the query is completed."""

        if user:
            stored_password = user['password']
            password = self.ui.passwordInput.text().strip()

            if self.user_repo.check_password(stored_password, password):
                try:
                    # Create a new session
                    self.session_id = self.session_repo.create_session(user["id"])
                    print(f"Session started: {self.session_id}")
                except Exception as e:
                    QMessageBox.critical(self, title: "Session Error", text: f"Failed to create session: {str(e)}")
                    return

                self.hide()
                self.new_window = MainWindow(user["id"], self.session_id)
                self.new_window.show()
            else:
                QMessageBox.warning(self, title: "Login", text: "Invalid password!")
        else:
            QMessageBox.warning(self, title: "Login", text: "User not found!")
```

Figură 18 Exemplu de clasă controller pentru pagina de login

Figura de mai sus reprezintă clasa **controller** pentru pagina de logare a unui utilizator. Această clasă folosește clasa generată de **PySide6** (clasă generată pe baza fișierelor .ui create în aplicația Desginer). Această clasă se ocupă cu procesul de logare a unui utilizator. După ce utilizatorul completează datele de logare și apasă butonul de logare, funcția de logare din repository este folosită pentru a verifica credențialele utilizatorului respectiv.



Figură 19 Pagina de logare a unui utilizator în aplicația TOPIKKA

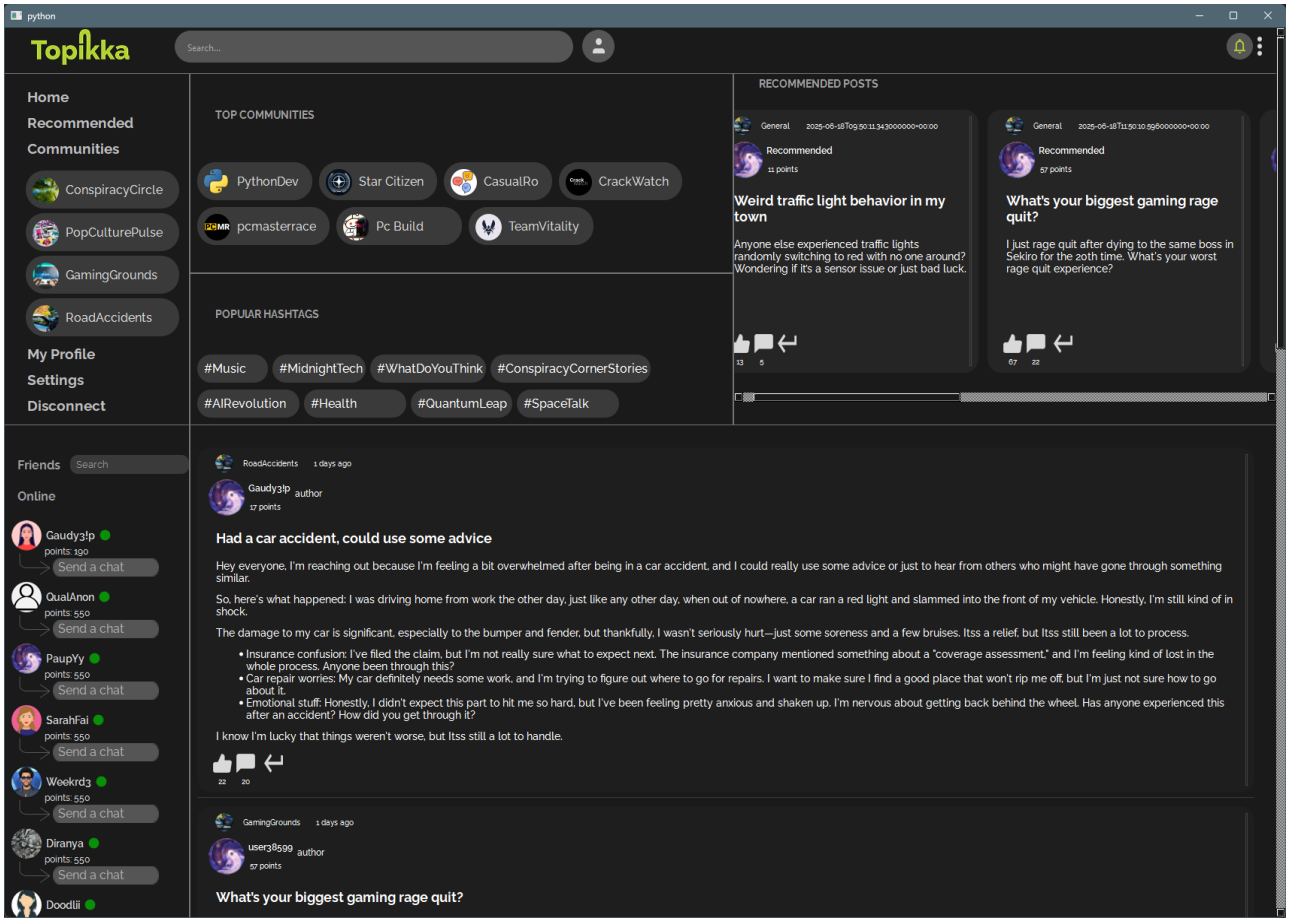
Figura de mai sus reprezintă pagina de logare din aplicația TOPIKKA care utilizează clasa precizată anterior care se ocupă cu logica de logare. Acest proces de *clasă controller* reprezentând logica interfeței și care folosește o clasă **generată** pentru a fi utilizată ca design de bază se regăsește la restul claselor din modulul **frontend.**

Un ultim exemplu de o pagină din interfață este cea de **Dashboard** care reprezintă interfața principală după ce utilizatorul s-a logat cu succes în aplicație. Clasa de tip controller **dashboardWindow.py** care conține următoarele funcții:

- create_feed_container: Crează containerul care conține lista de widget-uri de tip postări afișate în feed-ul principal.
- load_posts: gestionează logica de încărcare a listei de postări primite din clasa **PostRepository** în interfață
- on_posts_error: gestionează eventualele erori.
- show_post_detail: logica de comutare a interfeței în modul de vizualizare a unei postări în detaliu.
- handle_comment, handle_reply: funcții care gestionează logica butoanelor.

- load_communities, load_friends, load_tags și load_recommended_posts: funcții care gestionează toate datele necesare de populare a interfeței.

Următoarea figură reprezintă rezultatul final al utilizării funcțiilor de mai sus.



Figură 20 Interfața DashBoard din aplicația TOPIKKA

CONCLUZII

În concluzie, scopul acestei teme a fost dezvoltarea unei aplicații folosind limbajul Python împreună cu o bază de date de tip graf, iar acesta fiind atins prin crearea aplicației prezentate. Pe baza capitolului 2 și 3 putem deduce că utilizarea unei baze de date de tip graf pentru o aplicație socială este o alegeră bună datorită naturii acestei aplicații și a obiectivelor stabilite.

În prezent bazele de date de tip graf evoluează foarte mult, fiind utilizate de diverse companii mari precum Microsoft (AWS și Microsoft Azure), Google Cloud, snowflake și databricks. Această tehnologie este și va fi populară în continuare datorită oportunităților pe care le oferă, un exemplu foarte bun este implementarea AI-ului în baze de date de tip graf, unde inteligența artificială a avut un boom de popularite în ultimii ani iar prin utilizarea limbajului Python care proficient în machine learning și AI este un limbaj perfect pentru a fi folosit împreună cu bazele de date de tip graf.

Din punct de vedere a limitelor unei baze de date de tip graf cele mai întâlnite sunt Scalabilitatea orizontală limitată, standardizarea redusă a limbajelor de interogare, integrarea dificilă în ecosisteme tradiționale și costuri ridicate oferite de soluții comerciale. Aceste limitări sunt greu de depășit deoarece necesită o evoluție radicală și majoră în domeniul bazelor de date de tip graf. Luând în vedere aceste limitări bazele de date încep să fie din ce în ce mai folosite în diverse domenii unde oferă performanțe mai bune față de o bază de date tradițională, cel mai mai populare exemple fiind în rețele sociale și hărți[11].

În opinia mea, utilizarea unei baze de date de tip graf este una benefică, având în vedere oportunitățile de a crea relații complexe între entități și flexibilitatea ridicată pentru dezvoltarea unei baze de date complexe. Pentru o aplicație de tip rețea socială, utilizarea acestei baze de date este ideală, oferă posibilitatea de a implementa relații complexe și de a utiliza diverși algoritmi de recomandări. Însă există și dezavantaje care trebuie luate în calcul, costurile pentru întreținerea unei baze de date de tip graf sunt mult mai mari față de baze de date tradiționale și complexitatea acestei baze de date poate deveni un dezavantaj pentru aplicații sau sisteme de dimensiuni mari.

[11] https://neo4j.com/docs/getting-started/graph-database/#limitations

Bibliografie

Amy E. Hodler, M. N. (2022). *Graph Data Science Using Neo4j*. Taylor & Francis Group.

Beazley, D., & Jones, B. (2013). *Python Cookbook, Third Edition.* O'Reilly Media.

Bruggen, R. V. (2014). *Learning Neo4j.* Packt Publishing.

Chang, V. S. (2022). *Scientific Data Analysis using Neo4j* . Science and Technology Publications.

Disney, A. (2023, October 19). *Cambridge Intelligence*. Preluat de pe Cambridge Intelligence: https://cambridge-intelligence.com/choosing-graph-database/

Fitzpatrick, M. (2025, May 19). *Which Python GUI library should you use?* Preluat de pe https://www.pythonguis.com/faq/which-python-gui-library/

Gupta, S. (2015). *Building Web Applications with Python and Neo4j: Develop exciting real-world Python-based web applications with Neo4j using frameworks such as Flask, Py2neo, and Django.* Packt Publishing.

Joshi, V. (2017, Mar 20). *A Gentle Introduction To Graph Theory*. Preluat de pe Medium: https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8

Kemper, C. (2015). *Beginning Neo4j*. Apress.

Learn, F. (2019). *Figma*. Preluat de pe What is Figma: https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma

Mal, S. (2023, August 15). *Linkedin*. Preluat de pe Linkedin: https://www.linkedin.com/pulse/graph-databases-future-scalable-flexible-data-complex-/

Martin Fowler, D. R. (2002). *Pattern of Enterprise Application Arhitecture.* Addison Wesley.

Mathematics | Graph Theory Basics - Set 1. (2025, Apr 14). Preluat de pe geeksforgeeks: https://www.geeksforgeeks.org/mathematics-graph-theory-basics-set-1/

McKinney, W. (2022). *Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter*. O'Reilly Media.

Needham, M., & Hodler, A. (2019). *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media.

Neo4j. (2021). *Cypher Cheat Sheet*. Preluat de pe Neo4j: https://neo4j.com/docs/cypher-cheat-sheet/5/all/

Neo4j. (2021). *Neo4j Breaks Scale Barrier with Trillion+ Relationship Graph*. Preluat de pe Neo4j: https://neo4j.com/press-releases/neo4j-scales-trillion-plus-relationship-graph/

Neo4j. (fără an). *Neo4j Docs*. Preluat de pe Neo4j: https://neo4j.com/docs/getting-started/graph-database/#limitations

Neo4j, I. (2021). *Cypher Query Language Reference, Version 9.* Neo4j, Documentație oficială.

Palifka, B. (2024, September 27). *Linkedin*. Preluat de pe Why Multi-Hop Queries Are Easier in a Graph Database: https://www.linkedin.com/pulse/why-multi-hop-queries-easier-graph-database-bill-palifka-0l2oe/

Panzarino, O. (2014). *Learning Cypher.* Packt Publishing.

Postgres. (fără an). *PostgreSQL 17 Released*. Preluat de pe PostgreSQL : https://www.postgresql.org/about/news/postgresql-17-released-2936/

Python Documentation. (2001). Preluat de pe Python: https://www.python.org/doc/essays/blurb/

Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming.* O'Reilly Media.

Rozemberczki, B. (2019). *MUSAE GitHub Social Network.* Preluat de pe Kaggle: https://www.kaggle.com/datasets/rozemberczki/musae-github-social-network

Scifo, E. (2023). *Graph Data Science with Neo4j: Learn how to use Neo4j 5 with Graph Data Science library 2.0 and its Python driver for your project.* Packt Publishing.

See Shields, R. (2012). https://ro.wikipedia.org/wiki/Problema_podurilor_din_K%C3%B6nigsberg. *Problema podurilor din Königsberg*, 1.

Tahir, N. (2023, Mar 8). *Cypher Query VS SQL*. Preluat de pe Medium: https://medium.com/@nimratahir1212/cypher-query-vs-sql-718ccb8e8d9d

University of San Francisco, S. F. (2020, 05 30). *Can Neo4j Replace PostgreSQL in Healthcare?* Preluat de pe National Library of Medicine PubMed Central: https://pmc.ncbi.nlm.nih.gov/articles/PMC7233060/#:~:text=Can%20Neo4j%20Replace%20PostgreSQL%20in,exception%20of%20metadata%20queries%2C