

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Gemixque

sistem de recomandări de jocuri video

propusă de

Radu Damian

Sesiunea: *iunie/iulie, 2022*

Coordonator științific

Lect.dr. Cristian Frăsinaru

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Gemixque

sistem de recomandări de jocuri video

Radu Damian

Sesiunea: *iunie/iulie, 2022*

Coordonator științific

Lect.dr. Cristian Frăsinaru

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și Prenume _____

Data _____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemnatul(a) _____
domiciliul în _____
născut(ă) la data de _____, identificat prin CNP _____,
absolvent(a) al(a) Universității "Alexandru Ioan Cuza" din Iași,
Facultatea de _____ specializarea _____,
promoția _____, declar pe propria răspundere, cunoscând consecințele
falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației
Naționale nr. 1/2011 art.143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul:

elaborată sub îndrumarea dl. / d-na _____
pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum
conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată
prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la in-
troducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări
științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei
lucrări de licență, de diplomă sau de disertație în acest sens, declar pe proprie răspundere
că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data azi, _____

Semnătură student _____

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul "*Gemixque*", codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea "Alexandru Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Radu Damian*

(semnătura în original)

Mulțumiri

Sunt profund recunoscător pentru sprijinul necondiționat oferit de părinții și de bunica mea în vederea realizării acestei lucrări de licență.

Îi mulțumesc Monicăi pentru faptul că m-a ajutat să-mi gestionez trăirile și să privesc provocările pe care le-am întâmpinat cu o atitudine sănătoasă.

De asemenea, îi mulțumesc domnului prof. dr Frăsinaru Cristian, îndrumător al lucrării de licență, pentru sfaturile, resursele, și încrederea oferită de-a lungul perioadei în care am lucrat pentru licență.

Cuprins

1	Descrierea problemei	5
1.1	Scopuri și cerințe ale aplicației	5
2	Tehnologii folosite	6
3	Arhitectura sistemului	7
4	Baza de date	8
4.1	Introducere	8
4.2	De ce NoSQL și nu SQL?	9
4.3	Modelarea bazei de date	11
4.4	Potențiale probleme	14
4.5	Generarea datelor de intrare	16
4.6	Importarea conținutului fișierelor CSV în baza de date	21
5	Back-end	24
5.1	De ce Java? De ce Spring Boot?	24
5.2	Modelarea nodurilor în entități	24
5.3	Interfața Repository și conceptul de Interface Projection	26
5.4	REST API	27
5.5	Modulul de autentificare al utilizatorilor	28
6	Algoritmul de recomandare	31
6.1	Introducere	31
7	Front-end	31
8	Manual de utilizare	31
9	Concluzii și direcții viitoare	31

Introducere

Această lucrare de licență propune prezentarea implementării unui sistem de recomandare, sub forma unei aplicații web.

În următoarele capitole vor fi ilustrate următoarele: problema ce trebuie rezolvată, tehnologiile folosite, modelarea bazei de date, modul în care au fost generate datele de intrare ale problemei, server-ul de back-end, algoritmul implementat și server-ul de front-end.

Motivație

Ideea de bază a lucrării de licență a venit oarecum din întâmplare. Când eram student în semestrul II din anul 2 de facultate, mă uitasem într-o zi obișnuită la acest site: <https://simkl.com>, pe care-l foloseam pentru a nu uita la ce episod am rămas dintr-un anumit serial. Acest site poate oferi și recomandări de filme/seriale în funcție de datele din profilul meu.

Fiind un amator de jocuri video pe PC, mi-a venit ideea de a face un astfel de site, însă care să recomande, bineînțeles, jocuri video.

O observație importantă este faptul că aş fi putut alege orice resursă ce poate fi recomandată (de exemplu: periute de dinți, telefoane, picturi etc.), aşadar problema în speță poate fi privită într-un mod mai general. Motivul pentru care am ales jocurile video este dat de pasiunea mea pentru acestea.

De asemenea, eram curios în acea perioadă, printre altele, legat de cum aş putea implementa o soluție pentru acest sistem de recomandare pe care mi l-am propus să-l realizez.

Aşadar, am avut inspirație, dar și noroc, deoarece această idee mi-a venit relativ repede, lucru ce a reprezentat un punct de pornire important în realizarea efectivă a acestei lucrări.

Aplicații similare

Din aplicațiile similare căutate, am identificat următoarele pe care le-am considerat mai notabile:

- IGDB - <https://www.igdb.com/>
- Steam - <https://store.steampowered.com/>
- Metacritic - <https://www.metacritic.com/>
- Quantic Foundry - <https://quanticfoundry.com/>
- Games finder - <https://gameslikefinder.com/>

Încercând să caut informații legate de modul concret de implementare al recomandărilor care se regăsesc în aceste aplicații, mi-am dat seama că acest lucru este în zadar, din cauza aspectului comercial al aplicațiilor.

Chiar și așa, aplicația ce va fi prezentată în această lucrare de licență are un algoritm de recomandare ce a necesitat o documentare în prealabil, și utilizează de asemenea o bază de date nerelațională de tip graf.

Așadar, aplicația propusă s-ar putea descrie mai degrabă ca o tentativă de a îmi răspunde la următoarea întrebare care nu mi-a fost răspunsă consultând aplicațiile similare: Cum se poate implementa un algoritm de recomandare?

Bineînțeles că răspunsul nu este unul universal, ci propun o variantă (din nenumăratele care pot exista, căci este loc de inovații și contribuții semnificative în a ataca această problemă) care a fost șlefuită folosind resurse proprii.

În timpul implementării aplicației, mi-am dat seama că răspunsul meu poate fi privit doar ca un punct de pornire pentru a dezvolta în continuare partea de recomandare a aplicației. Aceste reflecții pot fi regăsite în capitolul **Concluzii și direcții viitoare**.

1 Descrierea problemei

Problema ce trebuie rezolvată se rezumă la recomandarea unor jocuri video(niciunul, unul, sau mai multe) unui utilizator. În ansamblu, datele de intrare ar putea fi reprezentate de o mulțime de jocuri video, o mulțime de utilizatori și o mulțime de recenzii.

Aceste recenzii sunt oferite de utilizator și asociate unui joc. În recenzie, utilizatorul oferă o notă(un scor, pe o scară de la 1 la 10) care indică cât de mult i-a plăcut/displăcut jocul respectiv. Un utilizator poate face maxim o recenzie per joc.

O descriere mai amănunțită a utilizatorilor, recenziilor și a jocurilor se poate regăsi în capitolul 4 **Baza de date**, subcapitolul **Modelarea bazei de date**.

1.1 Scopuri și cerințe ale aplicației

Scopul documentului

Scopul acestui document este de a ilustra modul în care funcționează un sistem de recomandări de jocuri video sub forma unei aplicații web având ca soluție de stocare o bază de date de tip graf NoSql.

Publicul țintă

Acest document este destinat atât cititorilor avizați(e.g. profesori universitari) pentru a afla de exemplu soluțiile utilizate în cadrul capitolelor 4, 5 sau 6, **Baza de date**, **Back-end** respectiv **Algoritmul de recomandare**, cât și cititorilor neavizați(utilizatori obișnuiți), care se pot informa în legătură cu modul în care pot interacționa cu site-ul în capitolul 8 **Manual de utilizare**.

Scopul aplicației

Scopul acestei aplicații este de a oferi o soluție în a contracara cantitatea masivă de informații ce se regăsește pe internet [5] în ceea ce privește multitudinea de jocuri video, prin a dezvolta un sistem de recomandări care să faciliteze decizia unui utilizator legat de ce joc să aleagă.

Contribuții

Generarea datelor de intrare ale problemei a fost realizată de mine, în sensul că nu am folosit în mod direct seturi de date externe(de exemplu: Kaggle), ci am implementat un modul în Java care generează fișiere tip csv.

De asemenea, implementarea algoritmului a fost făcută fără utilizarea unei librării externe(cum s-ar fi putut face de exemplu cu Python). Motivația acestei decizii este reprezentată de a avea o perspectivă mai profundă a unui posibil algoritm de recomandare.

2 Tehnologii folosite

Pentru aplicația web:

1. Baza de date: Neo4j.
2. Server de back-end:
 - Limbaj: Java versiunea 11
 - Framework: Spring Boot
 - REST API
3. Server de front-end:
 - Limbaj: Typescript
 - Framework: Angular

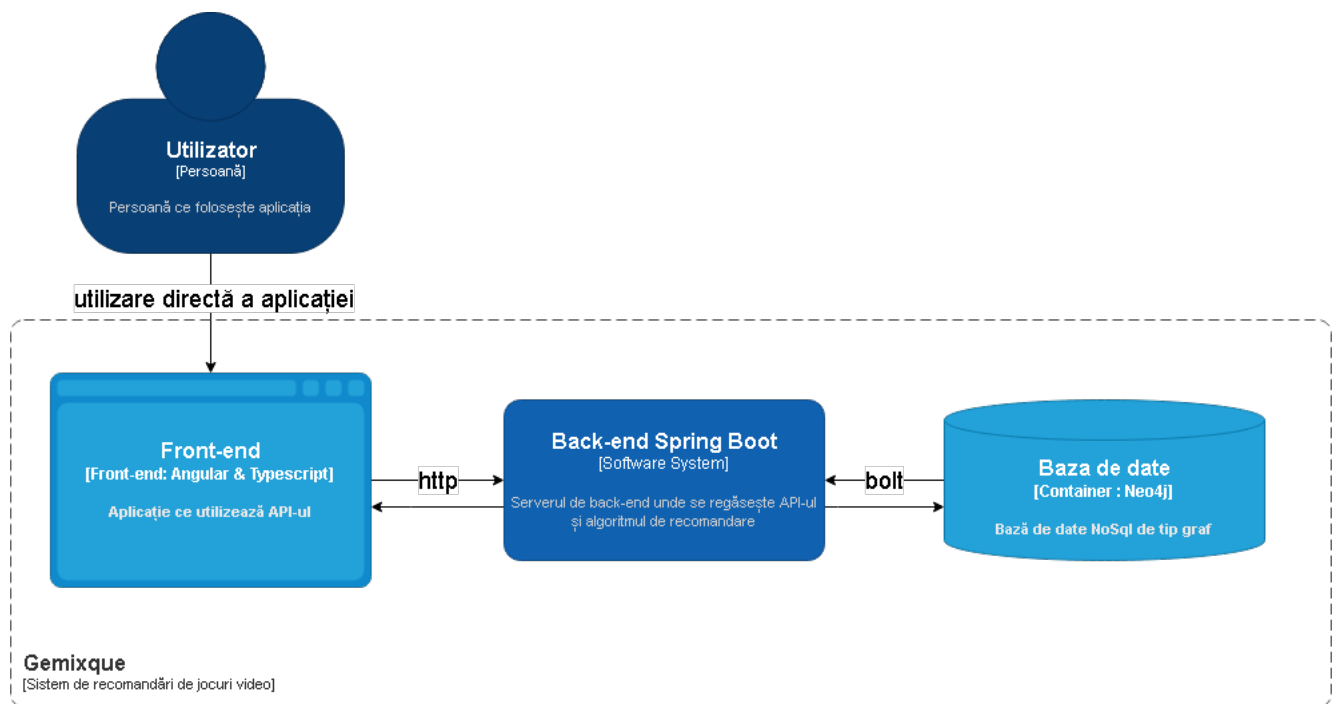
Pentru modulul de generare al datelor de intrare:

1. API extern folosit pentru a procura date despre jocuri video: IGDB API [1]
2. API extern folosit pentru procurarea de recenzii de pe Steam [2]
3. Java Faker - pentru generarea de date aleatoare [3]
4. Apache Commons CSV - pentru manipularea de fișiere CSV [4]

3 Arhitectura sistemului

Arhitectura sistemului ce propune rezolvarea problemei este următoarea:

Figure 1



Diagramă C4 nivelul 1

Se poate observa în ansamblu suita de tehnologii folosite: Neo4j pentru baza de date, Spring Boot pentru back-end și Angular pentru front-end.

Schema bazei de date va fi reprezentată în capitolul următor, deoarece înțelegerea acestuia presupune o scurtă inițiere în ceea ce propune Neo4j.

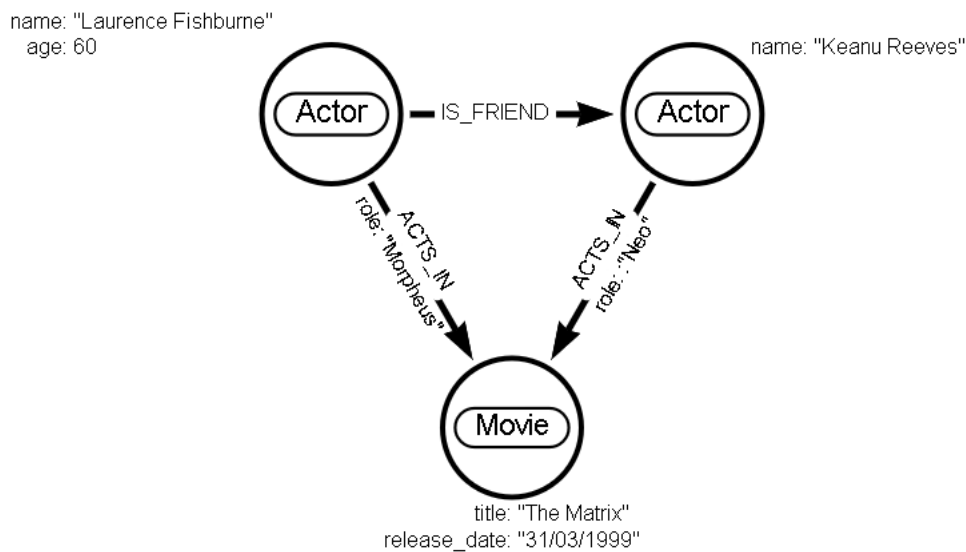
4 Baza de date

4.1 Introducere

Tipul de stocare ales este cel oferit de Neo4j, în care se prezintă o abordare NoSQL de tip graf. Comparativ cu o bază de date relațională, în care datele sunt stocate prin intermediul unor înregistrări(tuple) în tabele, în Neo4j datele sunt stocate prin intermediul nodurilor și muchiilor.

În exemplul următor, vor fi ilustrate caracteristicile nodurilor și muchiilor în stocarea efectivă a datelor:

Figure 2

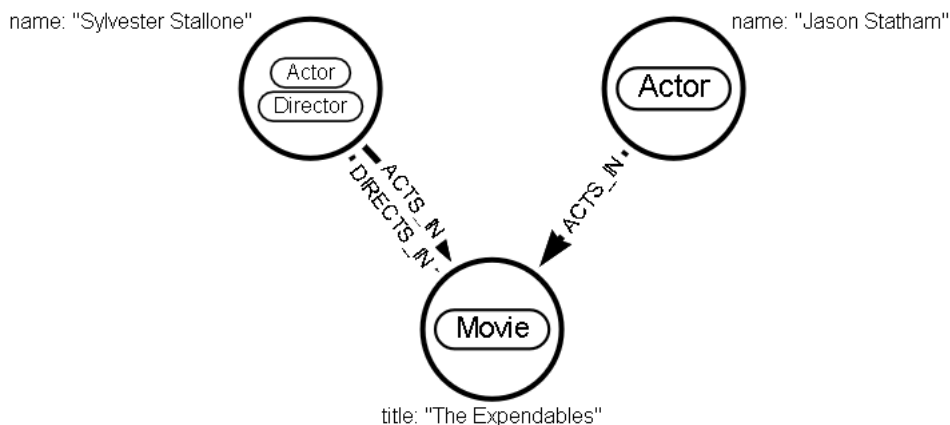


Un nod în neo4j are următoarele caracteristici [6]: reprezintă entități/obiecte, pot fi etichetate și pot avea proprietăți.

Observăm din figură cele trei entități: două noduri etichete cu 'Actor' și un nod etichetat cu 'Movie'. De asemenea, ambele noduri 'Actor' au proprietatea 'name', însă doar unul din ele are și proprietatea 'age'. Așadar, nu trebuie neapărat ca două noduri cu aceeași etichetă să aibă aceleași proprietăți.

Încă un lucru important de menționat este faptul că un nod poate avea mai multe etichete, așa cum se poate vedea din următorul exemplu:

Figure 3



Practic, din această figură se observă faptul că se poate modela cu ușurință situația în care un regizor joacă în propriul său film.

O altă noțiune importantă într-o bază de date de tip graf este cea de relație, care este asociată unei muchii. O relație trebuie să aibă un tip, un sens, și poate avea proprietăți. Din figurile anterioare s-au putut observa relațiile **ACTS_IN**, **DIRECTS_IN** sau **IS_FRIEND**.

Un alt aspect important de precizat este faptul că între două entități pot exista mai multe relații, așa cum s-a putut vedea în figura anterioară. Așadar, se modelează practic un multigraf orientat.

4.2 De ce NoSQL și nu SQL?

Având în vedere faptul că în această aplicație este prezentă o rețea socială în care mai mulți utilizatori pot interacționa între ei și pot oferi recenzii jocurilor video, o bază de date de tip graf este o alegere inspirată.

În ceea ce privește limbajul utilizat pentru a efectua interogări pe baza de date, avem în vedere următorul studiu de caz:

Să presupunem că dorim să modelăm ceea ce se întâmplă în cadrul unei facultăți, pe scurt: gestionarea studenților, a notelor pe care le iau aceștia la cursuri, și a profesorilor. Exemplul ce urmează a fi ilustrat se bazează pe schema bazei de date ce a fost utilizată în cadrul materiei Baze de date din anul II semestrul I. [7]

Să propunem că dorim să efectuăm următoarea interogare: pentru un student, să aflăm numele profesorilor la cursurile în care studentul a luat nota 10.

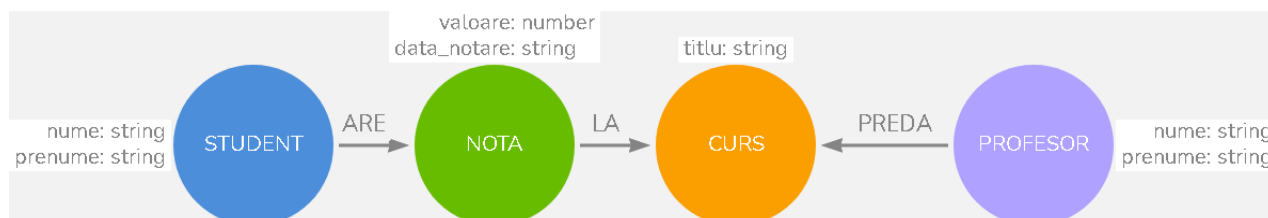
O interogare în limbajul SQL ar putea arăta în felul următor:

```
SELECT p.nume, p.prenume FROM NOTE n
JOIN CURSURI c ON n.id_curs = c.id
JOIN DIDACTIC d ON d.id_curs = c.id
JOIN PROFESORI p ON p.id = d.id_profesor
WHERE VALOARE = 10 AND ID_STUDENT = 36;
```

Se poate observa așadar faptul că sunt necesare o serie de join-uri pentru a putea obține rezultatul dorit.

Pentru a putea compara această interogare cu cea care s-ar putea face în limbajul Cypher folosit în Neo4j, se va ilustra pe scurt cum s-ar putea modela schema bazei de date anterior menționată în una de tip graf. (nu în totalitate, ci doar de ceea ce avem nevoie pentru a evidenția interogarea)

Figure 4: Miniatură a schemei bazei de date



Așadar, având în vedere acest model, interogarea în Cypher ar putea arăta în felul următor:

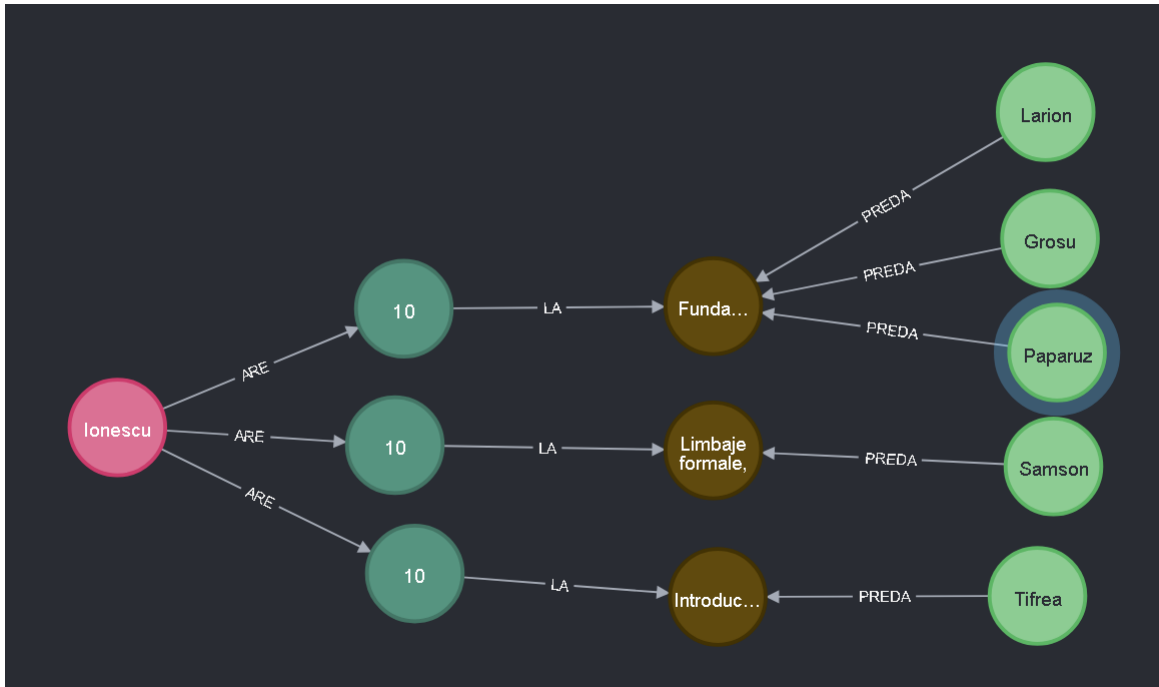
```
MATCH (s:STUDENT)-[:ARE]->(n:NOTA {valoare: 10}),  
(n)-[:LA]->(c:CURS)<-[:PREDA]-(p:PROFESOR)  
WHERE id(s) = 0  
RETURN p.nume, p.prenume
```

Un prim lucru interesant care s-ar putea observa este faptul că Cypher introduce prin sintaxa sa conceptul de ASCII Art [8], [9], care practic face posibilă o nouă interpretare a codului, una vizuală, adică se poate observa relativ ușor cum ceea ce este scris în primele două linii de cod seamănă destul de mult cu modelul din figura anterioară. Cele două linii s-ar fi putut scrie într-o singură linie fără nicio dificultate, însă am optat pentru această variantă pentru o mai bună lizibilitate a codului.

Pe de altă parte, o altă caracteristică importantă a bazelor de date de tip graf este că relațiile între entități au cea mai mare prioritate în modelarea datelor [10], acest lucru reflectându-se în modul cum a fost construită această interogare, în care este mult mai facil să interogăm niște date aflate la capătul unui lanț format din mai multe relații, așa cum se observă și în acest exemplu.

Pentru o mai bună înțelegere a modului în care sunt reprezentate efectiv datele, se poate vedea figura de mai jos:

Figure 5



Datele folosite pentru a exemplifica interogarea în Cypher

4.3 Modelarea bazei de date

Având în vedere noțiunile prezentate în capitolele anterioare, în acest capitol va fi descrisă schema bazei de date ce va fi folosită pentru a stoca datele necesare rezolvării problemei de a recomanda jocuri video.

Însă, înainte de a ilustra schema în ansamblu, fiecare entitate în parte va fi detaliată în rândurile următoare.

Utilizatorul

Această entitate reprezintă punctul central al aplicației, deoarece utilizatorul este cel care oferă recenzii(note) jocurilor și influențează într-un mod indirect recomandările unui alt utilizator, deci el este cel care, practic, inițiază acțiunea.

Nodul corespunzător acestei entități este cel din **Figura 5**.

Printre proprietățile alese pentru a reprezenta un utilizator, două dintre acestea reflectă activitatea lui în sistemul de recomandare, anume *average_score* și *nr_reviews_made*, care semnifică numărul total de recenzii făcute de utilizator, respectiv scorul mediu al acestuia(media tuturor scorurilor oferite).

Recenzia

Recenzia reprezintă practic indicatorul calitativ pe care-l oferă un utilizator asupra unui joc. Așa cum se observă din **Figura 6**, o recenzie poate fi caracterizată prin proprietatea *score*, nota oferită de utilizator jocului, proprietatea *content*, care reprezintă un

text ce poate fi introdus de utilizator pentru a-și exprima, dacă dorește, opinia sa asupra jocului, și proprietatea *time*, adică momentul în care a făcut recenzia.

Jocul

Jocul indică resursa ce va fi recomandată de către utilizator și este punctul de interes al aplicației. Faptul că această entitate poate avea un număr relativ semnificativ de proprietăți reprezintă o provocare justificată, iar abordarea mea pentru a reprezenta această entitate se poate observa în **Figura 7**.

Pentru a nu suprasatura un nod cu multe proprietăți, am decis să împart acestea în funcție de categoria în care ar putea fi încadrate. Nodul principal **Game** are doar titlul, genurile și anul primei lansări, iar prin relația de tipul **HAS** se evidențiază nodurile care extind această entitate, și anume cel care conține proprietățile care indică elemente vizuale ale jocului (*Visuals*), cel care conține date suplimentare despre joc (*Details*) și cel care ține evidența legat de recenziile aplicate asupra jocului (*Average*), unde proprietatea *value* reprezintă scorul mediu al jocului (cât de bine este notat în medie de către utilizatorii sistemului), iar proprietatea *aggregated_rating* reprezintă scorul obținut dintr-o sursă externă (e.g. Metacritic)

Schema bazei de date

În **Figura 8**, având în vedere și aspectele menționate anterior legat de entitățile care formează în ansamblu schema bazei de date, se poate intui fluxul principal al aplicației. Utilizatorul introduce în profilul său ce jocuri video s-a jucat prin intermediul relației **PLAYS**, iar acțiunea prin care acesta efectuează o recenzie asupra jocului este reprezentată prin relația de tip **MAKES**. De asemenea, relația **ON** indică asupra cărei entități etichetată cu *Game* se face recenzia.

Se poate deduce faptul că relațiile între noduri sunt numite astfel încât să reprezinte acțiuni ce pot fi efectuate sau aplicate asupra unor entități, ceea ce poate reprezenta un procedeu de bună practică în modelarea bazei de date în acest context. [11]

Figure 6: Utilizatorul

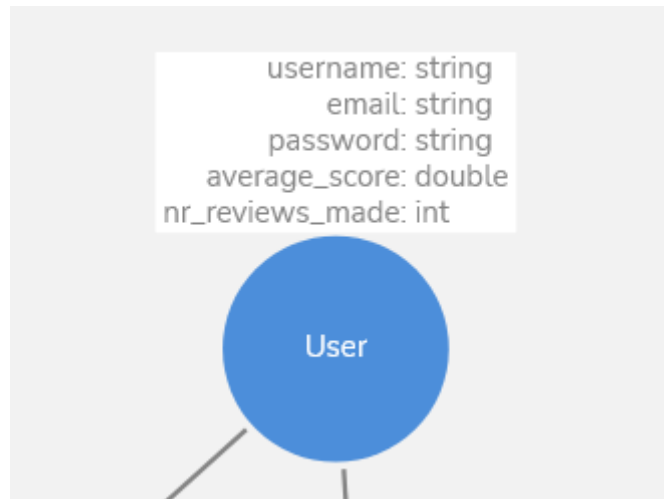


Figure 7: Recenzia

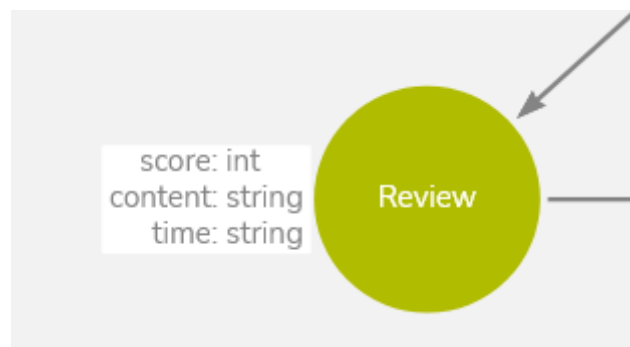


Figure 8: Jocul

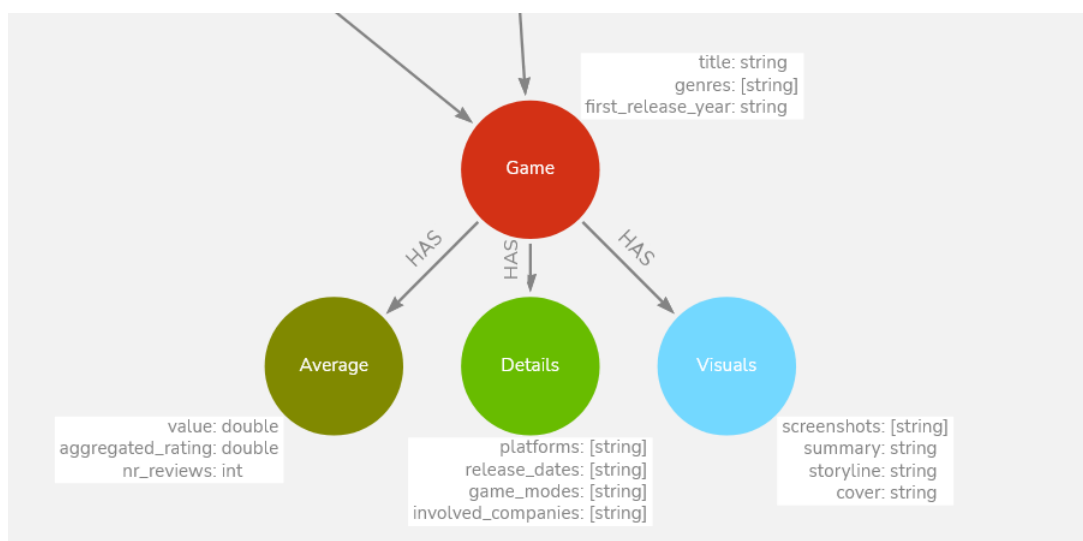
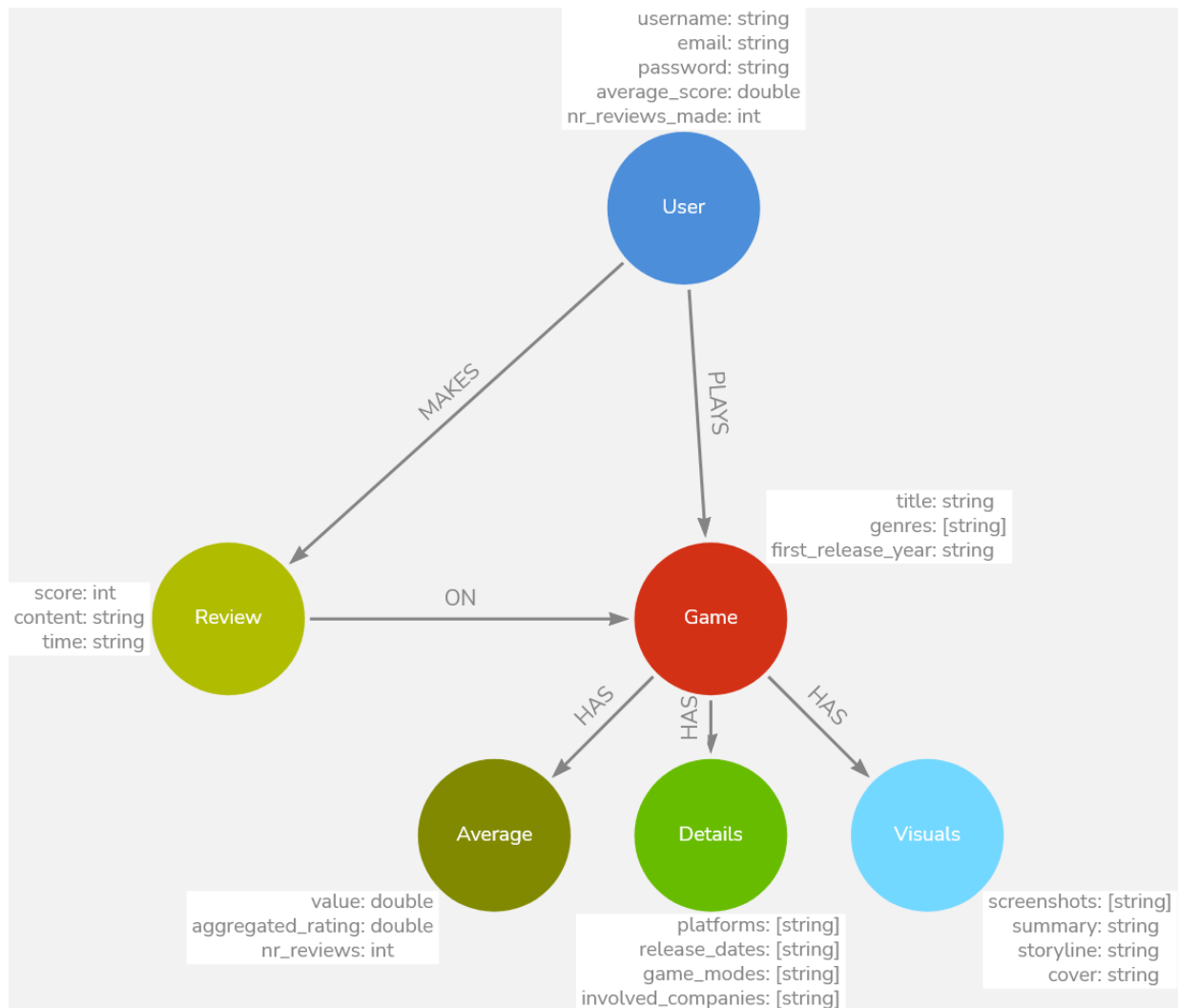


Figure 9: Schema bazei de date



Entitățile etichetate cu User, Game și Review mai au o proprietate în plus, și anume cea de id, care reprezintă un identificator unic pentru entitățile respective(uuid)

4.4 Potențiale probleme

Constrângeri

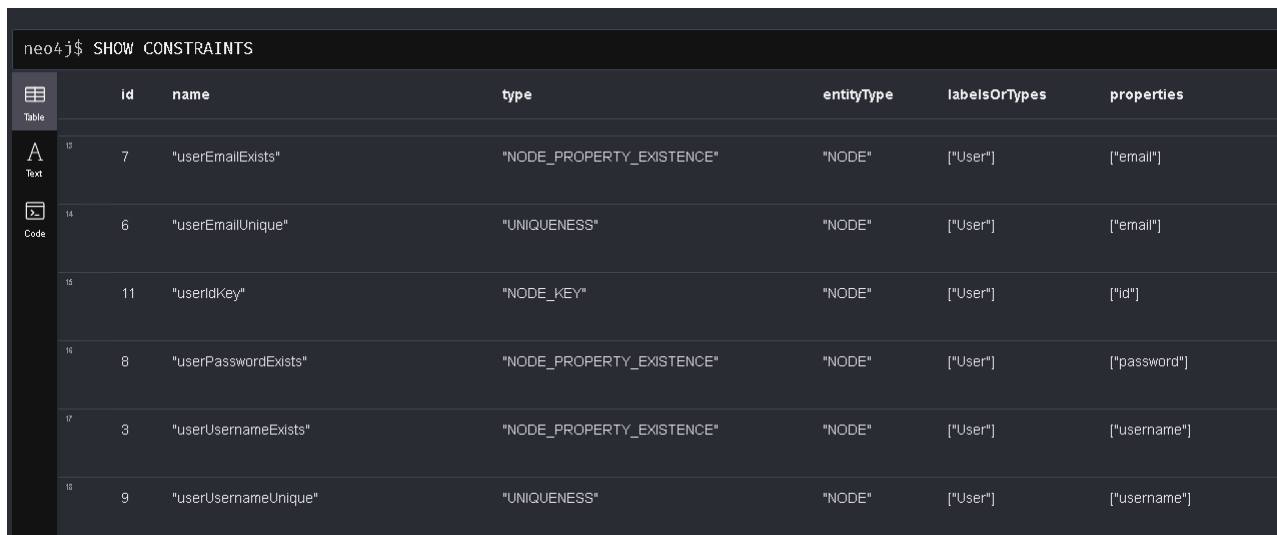
Am menționat atunci când am introdus capitolul de Baze de date faptul că două noduri care au aceeași etichetă nu trebuie să aibă neapărat aceleași proprietăți. Acest aspect face ca, în mod implicit, ca manipularea datelor să fie mult mai tolerabilă(cu mai puține constrângeri) în comparație cu o bază de date de tip SQL, unde acolo de exemplu trebuie să specificăm pentru o tabelă ce coloane să aibă și de ce tipuri să fie acestea.

Totuși, ce ar fi dacă am vrea să stabilim niște constrângeri? Poate am vrea să spunem de exemplu că o entitate etichetată cu User trebuie să aibă neapărat proprietățile *username*, *email* și *password*, sau am vrea să setăm pentru o entitate cheia sa, ca să o putem identifica. Pentru a rezolva acest lucru, Neo4j introduce conceptul de constrângeri (*constraints*) [12]

Pentru a lucra facil cu baza noastră de date, Neo4j oferă o aplicație desktop ce vine cu un panou de comandă care oferă multe utilități. Putem executa de exemplu interogări în Cypher și să vizualizăm în mod interactiv graful obținut, cum se poate observa și în **Figura 4**.

În figura de mai jos, am realizat o captură în acest panou de comandă în care am scris comanda `SHOW CONSTRAINTS` unde pot vedea ce constrângeri sunt setate.

Figure 10



	id	name	type	entityType	labelsOrTypes	properties
12	7	"userEmailExists"	"NODE_PROPERTY_EXISTENCE"	"NODE"	["User"]	["email"]
14	6	"userEmailUnique"	"UNIQUENESS"	"NODE"	["User"]	["email"]
15	11	"userIdKey"	"NODE_KEY"	"NODE"	["User"]	["id"]
16	8	"userPasswordExists"	"NODE_PROPERTY_EXISTENCE"	"NODE"	["User"]	["password"]
17	3	"usernameExists"	"NODE_PROPERTY_EXISTENCE"	"NODE"	["User"]	["username"]
18	9	"usernameUnique"	"UNIQUENESS"	"NODE"	["User"]	["username"]

Se pot observa constrângerile pe care le-am setat pentru entitatea User

Modul în care se actualizează scorul mediu al utilizatorului și al nodului *Average* asociat nodului *Game*

Am menționat atunci când am prezentat schema bazei de date de faptul că am dori să ținem cont de scorul mediu al utilizatorului și al jocului, și de numărul de recenzii al acestor entități. O variantă naivă pentru a rezolva această problemă este să calculăm valorile acestor proprietăți în serverul de back-end, parcurgând toate recenziile asociate unui utilizator/joc și calculând media aritmetică a scorurilor acestora de fiecare dată când se face o nouă recenzie de exemplu. Dacă avem relativ puține date, nu ar fi o problemă, însă atunci când numărul de recenzii, de jocuri sau de utilizatori crește semnificativ, atunci trebuie abordată o altă soluție pentru a eficientiza acest proces.

Neo4j introduce un alt concept, prin intermediul librăriei APOC [13], cel de declanșatoare (*triggers*). [14] Declanșatoarele sunt folosite pentru a executa cod în Cypher înainte sau după ce anumite date au fost modificate.

Așadar, am putea folosi un declanșator pentru a actualiza numărul de recenzii și scorul mediu pentru entitățile *User* și *Game* după ce o recenzie a fost creată.

Pentru a înțelege codul ce va fi prezentat ulterior, voi introduce un mic artificiu matematic pentru a înțelege cum se actualizează media scorurilor în acest caz.

Să presupunem că după k recenzii efectuate, scorul mediu este a_k și cea de a $k + 1$ recenzie are scorul s .

Pentru a reconstitui suma anterioară, aceasta reprezintă pur și simplu produsul dintre k și a_k . Așadar, putem calcula noua medie, prin formula:

$$a_{k+1} = \frac{k \cdot a_k + s}{k + 1}$$

Prin urmare, fiind introdusă această formulă, se poate prezenta codul ce va fi executat în cadrul declanșatorului, în rândurile următoare:

```
UNWIND \ $createdRelationships AS r
  MATCH (review:Review)-[r:ON]->(game:Game)
  MATCH (game)-[:HAS]->(average:Average)
  MATCH (u:User)-[:MAKES]->(review)-[:ON]->(game)
  SET average.value = toFloat(
    average.nr_reviews * average.value + toInteger(review.score)
  ) / (average.nr_reviews + 1),
    average.nr_reviews = average.nr_reviews + 1,
  u.average_score = toFloat(
    u.nr_reviews_made * u.average_score + toInteger(review.score)
  ) / (u.nr_reviews_made + 1),
  u.nr_reviews_made = u.nr_reviews_made + 1
```

A se ignora simbolul \ înainte de createdRelationships

Așadar, de fiecare dată după ce se creează o relație de tipul **ON** între nodurile *Review* și *Game*, atunci vom interoga nodurile *Average* și *User* pe care dorim să le actualizăm folosind formula anterioară.

4.5 Generarea datelor de intrare

Neo4j oferă funcționalitatea de a importa date prin intermediul fișierelor CSV, iar prin intermediul limbajului Cypher, pot fi create noduri și relații cu aceste date pentru a crea schema de baze de date.

Din intenția de a-mi modela baza de date după bunul meu plac, nu am vrut să optez pentru a lua un set de date deja gata făcut, ci am implementat un submodul în limbajul Java(separat de aplicația web propriu-zisă) care extrage date ori din API-uri externe(menționate la capitolul **Tehnologii și resurse folosite**, ori din fișiere salvate local, și generează fișiere CSV pentru a le putea importa apoi în baza de date.

Generarea utilizatorilor

Pentru a-mi genera utilizatorii fictivi ai aplicației, am apelat la librăria Java Faker. Aceasta m-a ajutat ca să generez nume de utilizator, parole și adrese de e-mail. În ceea ce privește parolele, am generat prin intermediul acestei librării un șir de caractere *random* format din 8 caractere, ce conține cifre și litere. De asemenea, pe această parolă am aplicat o funcție hash, cu scopul de a nu fi stocată parola originală în baza de date, deci pentru a avea un strat de securitate în acest caz. Am ales să utilizez algoritmul PBKDF2 în convertirea parolei deoarece este o variantă sigură. [15]

Așadar, cu valorile generate, am creat două fișiere CSV în care rețin 500 de utilizatori (un fișier cu parolele originale, celălalt cu parolele convertite prin funcția hash)

O observație importantă este că atât pentru utilizator, cât și pentru celelalte entități, am generat și un identificator unic, prin intermediul utilitarului UUID oferit de Java.

Generarea jocurilor

Pentru a-mi procura jocurile, m-am folosit de API-ul oferit de către cei de la IGDB.

Pentru a putea utiliza acest API, a fost necesară crearea unui cont pe Twitch, iar prin intermediul acestui cont, mi-am generat un *Client ID* și un *Client Secret*. Cu aceste credențiale, am generat un *token* pe care l-am folosit ca să pot accesa după API-ul menționat. Mai multe detalii pot fi regăsite aici. [16]

În figura următoare, am atașat o bucată de cod prin care se poate observa cum utilizez acest API pentru a procura jocurile.

Figure 11

```
private List<Game> getGames(){
    String FIELDS = "name,genres.name,aggregated_rating,platforms.name,release_dates.human,release_dates.platform.name,game_modes.name," +
        "involved_companies.company.name,screenshots.url,summary,storyline,cover.url,follows";
    APICalypse apicalypse = new APICalypse().fields(FIELDS).limit(300).sort( field: "follows", Sort.DESENDING).where("follows != null");
    try {
        return ProtoRequestKt.games(this.wrapper, apicalypse);
    } catch (RequestException e) {
        e.printStackTrace();
        System.out.println(e.getStatusCode());
        return null;
    }
}
```

Așadar, acele proprietăți enumerate în proprietatea FIELDS sunt practic cele folosite pentru modelarea entității *Game* din baza de date. Am procurat 300 de jocuri pe care le-am sortat descrescător după proprietatea *follows* pentru a mă asigura că voi lua cele mai populare jocuri cu putință.

După ce am obținut jocurile, am împărțit proprietățile așa cum am menționat la subcapitolul **Modelarea bazei de date** și am creat 4 fișiere CSV (games.csv, details.csv, visuals.csv, average.csv)

Algoritmul lui Durstenfield

Înainte de a trece la secțiunile următoare, am să prezint un algoritm care extrage în mod *random* un număr de k elemente dintr-o listă cu scopul de a-l folosi în repartizarea jocurilor și a recenziilor către fiecare utilizator.

Acest algoritm a fost introdus de către Richard Durstenfield și este o variantă modernizată a algoritmului Fisher-Yates. [16] Aceasta presupune parcurgerea listei (ori de la capăt, ori de la început) și schimbarea elementului curent cu un element ales în mod *random*. Dacă toată lista este parcursă, atunci vom obține o permutare a listei inițiale.

Deoarece avem de extras k elemente din listă, atunci nu e nevoie să parcurgem toată lista, ci doar primele/ultimele k elemente. După aceste k schimbări, putem extrage elementele dorite, ori de la începutul, ori de la sfârșitul listei.

O variantă de implementare a algoritmului în limbajul Java ar putea arăta în felul următor:

Figure 12

```
public class RandomUtil {

    public static <T> List<T> pickNRandomUsers(List<T> users, int n, Random r) {
        int length = users.size();
        for(int i = length - 1; i >= length - n; --i){
            Collections.swap(users, i, r.nextInt( bound: i+1));
        }
        return users.subList(length - n, length);
    }
}
```

Complexitatea acestui algoritm este $\mathcal{O}(n)$, cu condiția ca metoda care alege în mod întâmplător elementul care trebuie schimbat să se facă în complexitate $\mathcal{O}(1)$

Repartizarea jocurilor jucate către fiecare utilizator

În această secțiune se poate observa deja utilitatea algoritmului menționat anterior. Pentru fiecare utilizator, am ales la întâmplare 20 de jocuri din cele 300 generate, iar aceste asocieri le-am salvat în fișierul plays.csv

Acest fișier arată în felul următor:

Figure 13

	A	B	C	D	E
1	id	id_games_played			
2	f33ba2c8-9227-4327-be91-683ec7030c2b	[0f51573f-2b4d-4a11-86fd-e11cfaa03c66, c5fc			
3	d57088d2-cc5f-4a9d-904f-16c18d6c6b53	[7b26a3bd-0637-46d4-b24d-a6d3a4fa80df, d8			
4	5558c773-0c57-4c08-91ba-4d25ea189369	[ccda1837-717c-4d39-94cd-182678188114, e6f			
5	8b0ee7f6-012c-4f11-87cf-4233e77e7e2c	[1ff1b8fb-8057-49f4-b154-8c946f0f841b, a0b3			
6	0ff50b87-9012-4012-a330-afd4dd8d369a	[8942aa2f-3511-4d3c-87a2-204cadf87a4b, 7d8			
7	c2f04434-8e4b-4faf-b745-02f9dcc661c0	[e2191d78-ecc6-4d19-ac82-e2d8d00fb162, 29			
8	81daabe3-b9c8-4e39-86b1-29f28ce4f516	[0b501118-8e30-4b98-9332-ebbadbf5ef2d, 6a			
9	71a4fe5f-08c3-41ef-96a1-3d97e16a82e0	[f08f77e0-5a3a-4960-bdaa-f09effc3b269, 404			
10	9d7ad74c-dd68-4c18-84c4-7d3868faae25	[00dd7855-83a4-40d5-b2c1-8b24883a82ad, 7e			
11	de44971a-f53c-4248-a9eb-86f541e5e755	[62227181-03fc-49f3-bf88-0e48c10358d7, d16			
12	d7d873b0-d932-422f-b44e-a9c433e15e9b	[639091ef-23d6-4161-b702-efbc17577640, c9c			
13	50a4f192-522a-452e-a59a-ad2cc8aaf330	[29efe45b-5aaa-49af-8fe7-75c1d8ed73e9, 12			
14	e57180b8-a652-4e28-ade3-dfcbfae22691	[ca3e11c7-8386-4f18-a394-0f41cd5a931e, 86c			
15	d2c4b4c1-37cb-4d77-aff0-4fe13dfee963	[6853077e-1847-49d3-a384-fabf6fc3cc7a, ec6			

De menționat faptul că în coloana id_games_played sunt salvate niște liste de uuid-uri. În mod intuitiv, fișierele CSV generate ar putea fi asociate cu tabele dintr-o bază de date relațională. În acest caz, dacă am ignora faptul că id-urile jocurilor sunt reținute sub forma unei liste, am putea asocia conținutul fișierului plays.csv cu o tabelă asociativă. Practic aici se identifică o relație de mai mulți la mai mulți (un utilizator poate avea mai multe jocuri, iar un joc poate fi jucat de mai mulți utilizatori)

Generarea recenziilor și repartizarea acestora către fiecare utilizator

În această secțiune, am folosit atât API-ul oferit de Steam pentru a obține recenziile unui joc, dar și fișiere salvate local pentru a încerca să optimizez cât mai mult timpul necesar pentru a fi generate toate fișierele.

Primul pas a fost să salvez într-un fișier local în format JSON jocurile oferite de Steam, sub forma unei liste de obiecte, în care fiecare obiect are două proprietăți: *appid* (un identificator unic folosit de Steam pentru jocurile sale) și *name* (numele jocului). Scopul acestui fișier este de a încerca să identific dacă jocul procurat inițial de la IGDB API poate fi identificat ca un joc din platforma Steam ca să pot extrage ulterior recenzii relevante acestui joc. (prin intermediul API-ului oferit de Steam). În caz contrar, voi extrage o recenzie aleatoare dintr-un fișier salvat local de tip csv dintr-un set de date oferit de Kaggle. [17]

Al doilea pas a fost să asociez fiecărui utilizator un număr de recenzii pe care le poate oferi din cele 20 de jocuri pe care le-am repartizat. Inițial am extras din cele 20 de jocuri un număr din acestea stabilit în felul următor:

Voi genera un număr întreg în intervalul $[0, 5)$ cu probabilitate de 0.2, și cu probabilitate de 0.8 un număr în intervalul $[5, 20]$. Motivul pentru care am procedat astfel este ca să sporesc șansele ca doi utilizatori să aibă cât mai multe jocuri în comun la care au făcut recenzii. Am sporit aceste șanse pentru algoritmul de recomandare ce va fi descris într-un capitol ulterior.

În mod intuitiv, acest procedeu poate fi asociat prin alegerea unei urne din 2 disponibile. O urnă are probabilitate 0.2 de a fi aleasă, iar cealaltă urnă are probabilitate 0.8. În fiecare urnă, bilele (în cazul nostru numerele întregi) au aceeași probabilitate de a fi alese. (adică urmează o distribuție uniformă).

După ce am determinat acest număr, voi extrage acel număr de jocuri propriu-zis, după care voi încerca să asociez pentru fiecare joc câte o recenzie.

Pentru a fi mai ușor de explicat, următorii pași îi voi ilustra sub forma unui pseudocod, iar după acest pseudocod va fi prezentat codul propriu-zis în Java, pentru a fi mai ușor de înțeles.

Figure 14

```
For reviewedGame in reviewedGames {  
    review = '' //initialize review  
    if(isSteamGame(reviewedGame)) {  
        review = takeSteamReview(reviewedGame);  
    }  
    else {  
        review = takeRandomReview();  
    }  
    writeToCsv(review);  
}
```

La o primă vedere, acest procedeu nu este complicat, însă dificultatea constă în extragerea unei recenzii pentru un joc aflat pe platforma Steam, deoarece trebuie gestionată frecvența *request*-urilor care se fac către API-ul care ne oferă recenzii în acest caz. De menționat faptul că acest procedeu se realizează pentru fiecare utilizator.

O modalitate naivă este de a face un *request* de fiecare dată când dorim să asociem o recenzie unui joc, însă numărul de *request-uri* crește cu cât mărim numărul de utilizatori și de jocuri recenzate de fiecare utilizator. În cazul de față, avem 500 de utilizatori, iar în cel mai rău caz, pentru fiecare utilizator avem 20 de recenzii, deci în total 10000 de *request-uri*.

Să presupunem că timpul mediu în care s-ar face un request este de 20 milisecunde. (un timp foarte bun în practică) În total ar dura $10000 \cdot 20 = 200000 \approx 3$ minute, ceea ce este un timp deloc neglijabil în acest context.

O observație importantă este faptul că un joc căruia i se extrage o recenzie este foarte posibil să se repete atunci când se parcurg utilizatorii. De asemenea, nu suntem constrânși de către API să extragem fix o recenzie, putem extrage mai multe dacă se dorește acest lucru, ceea ce sporește varietatea recenziilor pentru același joc, în cazul în care se repetă.

O structură de date folositoare oferită de limbajul Java pentru a fructifica acest aspect este cea numită *Map*, în care se rețin perechi tip cheie-valoare. În contextul nostru, cheia este un joc, iar valoarea este reprezentată de recenziiile jocului, sub forma unui JSON. Inițializarea acestuia ar putea arăta în felul următor:

```
private final Map<SteamGame, JsonNode> reviewsRegistry = new HashMap<>();
```

Așadar, dacă jocul există deja în această structură de date, atunci voi extrage o recenzie deja stocată. În caz contrar, voi face un *request* pentru a obține recenziiile.

Codul implementat în Java arată în felul următor:

Figure 15

```

for (var reviewedGame : reviewedGames) {
    var maybeSteamGame : Optional<SteamGame> = steamGames
        .stream()
        .filter(steamGame -> steamGame
            .getName()
            .startsWith(reviewedGame.getName()))
        .sorted()
        .findFirst();

    String review = "";
    int score = ThreadLocalRandom.current().nextInt( origin: 1, bound: 11);
    String time = faker.date().past(faker.random().nextInt(1, 1000), TimeUnit.DAYS).toString();
    boolean takeRandom = true;
    if(maybeSteamGame.isPresent()){
        var steamGame : SteamGame = maybeSteamGame.get();
        this.updateReviewsRegistry(steamGame);
        if(reviewsRegistry.containsKey(steamGame)){
            var response : Pair<String, Boolean> = takeSteamReview(steamGame);
            review = response.getLeft();
            boolean votedUp = response.getRight();
            if(votedUp)
                score = ThreadLocalRandom.current().nextInt( origin: 7, bound: 11);
            else
                score = ThreadLocalRandom.current().nextInt( origin: 1, bound: 7);
            review = this.cleanReview(review);
            takeRandom = false;
        }
    }

    if(takeRandom){
        var record : CSVRecord = randomReviewsIterator.next();
        review = record.get("user_review");
    }
}

```

verific dacă jocul ce urmează a fi recenzat este un joc de pe platforma Steam

inițializare recenzie

prelucrez recenzia în cazul în care jocul se află în platforma Steam

extragere recenzie întâmplătoare

Se poate observa folosirea proprietății *votedUp*, pusă la dispoziție de către API-ul responsabil de recenzii de la Steam, prin care verific dacă acea recenzie a fost pozitivă sau negativă. În funcție de acest verdict, încerc să generez un scor care să reflecte cât mai mult realitatea.

Așadar, acesta este submodulul care se ocupă de generarea fișierelor CSV necesare pentru a putea importa conținutul acestora în baza de date.

4.6 Importarea conținutului fișierelor CSV în baza de date

Modul prin care sunt importate aceste date este prin intermediul limbajului Cypher, mai concret prin comanda `LOAD CSV` ce face parte din acest limbaj [18].

Un exemplu de cod în Cypher în care îmi creez recenziile generate poate fi acesta:

```

LOAD CSV WITH HEADERS
FROM 'file:///reviews.csv'
AS line
MATCH(u:User {id: line.id_user})
MATCH(g:Game {id: line.id_game})
MERGE(r:Review {id: line.id_review})
    ON CREATE SET r.score = line.score,
                r.content = line.content,
                r.time = line.time
MERGE (u)-[:MAKES]->(r)-[:ON]->(g)

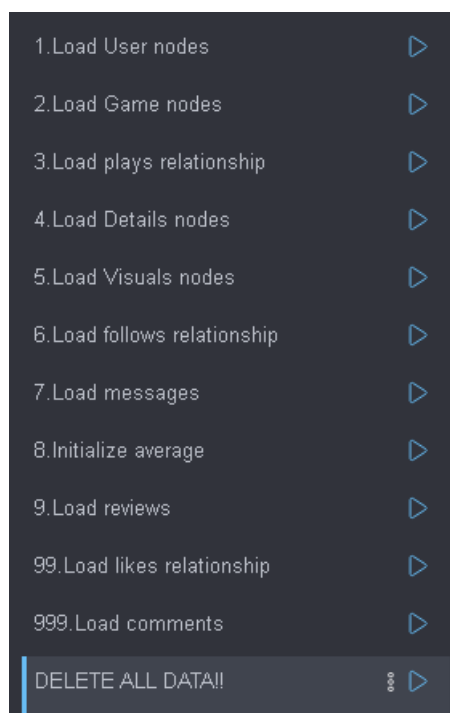
```

Un element de noutate este reprezentat de clauza `MERGE`. Prin aceasta putem verifica, în acest caz, dacă nodul *Review* a fost creat sau există deja. În cazul în care este creat, atunci vor fi setate proprietățile aferente. În cazul în care acest cod ar fi rulat a doua oară (fără alte modificări în prealabil), atunci nu va fi creat niciun nod nou, deci practic nu va mai avea niciun efect.

De-a lungul timpului cât am dezvoltat acest submodul de generare a fișierelor CSV, am întâmpinat uneori dificultăți care au făcut necesare ștergerea datelor din baza de date și reintroducerea acestora cu modificările aferente. Pentru a ușura acest proces, Neo4j oferă prin intermediul panoului de comandă o secțiune prin care se pot marca coduri în Cypher ca favorite și pot fi executate ori de câte ori este nevoie.

Practic, mi-am salvat o serie de coduri în secțiunea de favorite pe care le pot executa în ordine atunci când este nevoie.

Figure 16



În ceea ce privește ștergerea întregului conținut al bazei de date, aceasta trebuie făcută cu grijă. Neo4j recomandă mai multe alternative prin care acest procedeu poate fi făcut utilizându-se cele mai bune practici, pentru a evita efecte secundare neplăcute. [19]

Varianta pentru care am optat este următoarea:

```
CALL apoc.periodic.iterate(  
  "MATCH (n) return id(n) as id",  
  "MATCH (n) WHERE id(n) = id DETACH DELETE n",  
  {batchSize: 10000}  
)  
YIELD batches, total RETURN batches, total
```

Practic, prin intermediul librăriei APOC, șterg toate nodurile(cu tot cu relațiile asociate) în grupuri de câte 10000 de noduri.

După ce am importat datele, putem interacționa cu acestea prin intermediul panoului de comandă ca să se poată verifica dacă datele au fost întradevăr, introduse corect.

Una din modalitățile interesante prin care putem verifica asta este dacă schema bazei de date care a fost descrisă în subcapitolul **4.3 Modelarea bazei de date** se potrivește cu cea pe care o putem vizualiza în mod interactiv prin intermediul comenzii `CALL db.schema.visualization()`

Rezultatul acestei comenzi este reprezentat în figura de mai jos:

Figure 17



5 Back-end

5.1 De ce Java? De ce Spring Boot?

Soluția aleasă pentru a stoca datele este una relativ de nișă, având în vedere caracteristica de NoSql a bazei de date Neo4j, dar și de faptul că are la bază o reprezentare de tip graf. Așadar, studiind soluțiile oferite din această documentație [20] legată de limbajele suportate de Neo4j, am observat că Java este singurul limbaj în care Neo4j oferă capabilitatea de a mapa nodurile ce se regăsesc în baza de date în entități, ceea ce oferă un mod de lucru facil cu o perspectivă de ansamblu mult mai mare, în comparație cu un *driver* în care modul de lucru este unul mult mai elaborat. Această soluție se numește *Spring Data Neo4j*. [21]

Având în vedere faptul că capabilitățile de a lucra mult mai facil cu baza de date se regăsesc în suita Spring, am ales să utilizez Spring Boot pentru a implementa serverul de back-end. Faptul că am avut o introducere în acest *framework* în cadrul materiei *Programare avansată* din anul 2 semestrul 2 de studiu mi-a dat un punct de pornire consistent pentru a dezvolta această parte de back-end.

5.2 Modelarea nodurilor în entități

Așa cum am menționat la introducerea acestui capitol, soluția folosită pentru a mapa nodurile în entități este cea oferită de *Spring Data Neo4j*.

Această mapare presupune reprezentarea nodurilor din schema bazei de date (*User*, *Review* și *Game*) sub forma unor clase(entități) care pot fi utilizate pe partea de back-end. O instanță a acestei clase reprezintă practic un nod din baza de date.

La baza acestor mapări stau adnotările, iar cele care au fost utilizate în cadrul aplicației în mod frecvent sunt: *@Id*, *@Property*, *@Relationship*. Pentru a explica utilizarea lor, va fi reprezentată următoarea figură:

Figure 18

```

@Node("Game")
public class GameEntity implements Comparable<GameEntity> {

    @Id
    @GeneratedValue(UUIDStringGenerator.class)
    @Property("id")
    private String uuid;

    @Property
    private final String title;

    @Property
    private final List<String> genres;

    @Property("first_release_year")
    private final String firstReleaseYear;

    @Relationship(type = "HAS", direction = Relationship.Direction.OUTGOING)
    private VisualsEntity visuals;

    @Relationship(type = "HAS", direction = Relationship.Direction.OUTGOING)
    private DetailsEntity details;

    @Relationship(type = "HAS", direction = Relationship.Direction.OUTGOING)
    private AverageEntity average;

    @Relationship(type = "ON", direction = Relationship.Direction.INCOMING)
    private List<ReviewEntity> reviewsMade;
}

```

Entitatea *Game*

În primul rând, se observă adnotarea clasei *GameEntity* cu *@Node("Game")*, care asociază reprezentarea acestei clase cu cea a nodului din schema bazei de date etichetat cu "Game".

Cu adnotarea *@Id* se specifică proprietatea care identifică în mod unic o entitate. Având în vedere faptul că se utilizează *uuid*-uri, adnotarea *@GeneratedValue* ajută la specificarea unui generator care face în mod automat un *uuid*, un aspect util atunci când se instanțiază o nouă entitate, de exemplu.

Adnotarea *@Property* este utilă pentru a asocia proprietățile corespunzătoare nodurilor din schema bazei de date cu proprietățile din această clasă. (e.g. proprietățile *title*, *genres*, *firstReleaseYear*)

Adnotarea *@Relationship* este utilă atunci când se dorește ținerea în evidență a relațiilor pe care le poate avea entitatea definită cu celelalte entități. În acest caz, cum nodul *Game* are cele trei noduri *Visuals*, *Details* și *Average* care îl extind pe acesta, se poate observa în mod intuitiv utilizarea acestei adnotări. O relație este definită prin intermediul unui tip și al unei direcții ("înspre" entitatea definită, sau "în afara" entității.) De asemenea, se observă ținerea în evidență a recenziilor care s-au făcut asupra jocului.

O observație importantă în ceea ce privește încărcarea datelor asociate unei relații, este faptul că aceasta se face doar atunci când este în mod explicit nevoie. De exemplu, dacă am dori să obținem o entitate *GameEntity* în funcție de *uuid*, atunci datele obținute sunt doar cele aferente proprietăților (adică nu se vor încărca datele asociate nodurilor *Visuals*, *Details* și *Average*) pentru un plus de performanță.

Pentru a avea mai mult control asupra datelor ce se doresc a fi obținute, va fi prezentată secțiunea următoare.

5.3 Interfața Repository și conceptul de Interface Projection

Scopul interfeței Repository este de a oferi un mod abstract și facil prin intermediul căruia este implementat un strat de acces care poate fi utilizat pentru a reduce în mod semnificativ codul implementat în interacționarea cu datele. [22]

Această interfață este definită în mod generic, luând doi parametrii: numele entității care va fi prelucrată și tipul de date al identificatorului (în cazul de față *String*)

Metodele specifice acestei interfețe fac parte din repertoriul CRUD (Create, Read, Update, Delete.)

Un exemplu de definire a acestei entități poate fi observat în figura următoare:

Figure 19

```
public interface GameRepository extends Neo4jRepository<GameEntity, String> {
    List<GameInterfaceProjection> findAllProjectedBy();
    Optional<GameInterfaceProjection> findByUuid(String uuid);
    Page<GameInterfaceProjection> findAllProjectedBy(Pageable pageable);
}
```

Interfața de tip *Neo4jRepository* specifică entității *GameEntity*

În mod convențional, metodele sunt numite astfel încât să se poată observa tipul operației, folosindu-se cuvinte cheie precum *find*, *by*, *save*, iar implementarea acestora se face în mod automat la *runtime*. În exemplul din figură, metodele sunt de tipul Read: una pentru a obține toate jocurile stocate, una pentru a obține un joc după un *uuid*, și una pentru a obține jocurile prin intermediul paginării (de exemplu: o pagină de 5 jocuri)

Se oferă suport și pentru definirea unor metode *custom* în cazul în care este nevoie de obținerea unui rezultat în urma unei interogări în limbajul *Cypher*, însă mai multe detalii legate de acest aspect se pot regăsi în capitolul 6 **Algoritmul de recomandare**.

De asemenea, se poate observa apariția conceptului de *Interface Projection*, prin care se poate preciza prin intermediul unei interfețe ce date să fie prelucrate dintr-o entitate.

În cazul de față, să presupunem că dorim să avem pentru entitatea *GameEntity* și acele date care extind nodul *Game* aferent din baza de date (*Visuals*, *Details* și *Average*). Așadar, putem să ne definim o interfață (de exemplu cu numele de *GameInterfaceProjection*) care precizează datele de care avem nevoie. Aceasta arată în felul următor:

Figure 20

```

public interface GameInterfaceProjection {
    String getUuid();
    String getTitle();
    List<String> getGenres();
    VisualsInterfaceProjection getVisuals();
    DetailsInterfaceProjection getDetails();
    AverageInterfaceProjection getAverage();
}

```

Un lucru interesant de menționat este faptul că aceste proiecții pot fi definite în mod recursiv. În cazul de față, *Visuals*, *Details* și *Average* sunt de asemenea entități, și datele acestora pot fi controlate separat prin intermediul unui *Interface Projection* specific.

5.4 REST API

Modul prin care funcționalitățile de pe back-end sunt expuse spre a fi folosite de către serverul de front-end este prin intermediul unui REST API, în care resursele oferite sunt reprezentate sub format JSON.

Expunerea acestor funcționalități s-a făcut prin intermediul *Controller*-elor: *UserController*, *GameController* și *ReviewController*. Acestea utilizează în mod intern servicii implementate care se utilizează de metodele din *repository*. De exemplu, se poate regăsi în figura de mai jos modul cum a fost definit *ReviewController*:

Figure 21

```

@RestController
@RequestMapping("api")
public class ReviewController {

    @Autowired
    private ReviewService reviewService;

    @GetMapping("/reviews")
    public List<FullReviewInterfaceProjection> getAllReviews() { return reviewService.getAllReviews(); }

    @GetMapping("/reviews/{id}")
    public FullReviewInterfaceProjection getReview(@PathVariable String id){
        return reviewService.getReviewById(id).orElseThrow(ResourceNotFoundException::new);
    }

    @GetMapping("/users/{userId}/reviews")
    public GetReviewsMadeFromUserInterfaceProjection getReviewsFromUser(@PathVariable String userId){
        return reviewService.getReviewsMadeByUser(userId).orElseThrow(ResourceNotFoundException::new);
    }

    @GetMapping("/games/{gameId}/reviews")
    public GetReviewsMadeOnGameInterfaceProjection getReviewsOnGame(@PathVariable String gameId){
        return reviewService.getReviewsMadeOnGame(gameId).orElseThrow(ResourceNotFoundException::new);
    }
}

```

Adnotările sunt și în acest caz destul de sugestive. Practic, sunt definite mai multe *request-uri* de tipul GET ce pot fi accesate prin intermediul protocolului HTTP. Acestea au rolul de a oferi resurse(recenzii), mai concret ori toate recenziile disponibile, ori o recenzie în funcție de id, ori toate recenziile făcute de un utilizator, ori toate recenziile făcute asupra unui joc.

Adnotarea *@Autowired* denotă faptul că instanța serviciului *ReviewService* este injectată spre a fi folosită în acest controller. Acest aspect ilustrează conceptul de *Dependency Injection*, care are ca scop reducerea cuplării. [23]

Acest serviciu este responsabil de procurarea datelor necesare prin intermediul *repository-urilor* definite.

Modul cum a fost implementat se regăsește în următoarea figură:

Figure 22

```
@Service
public class ReviewService {

    @Autowired
    private ReviewRepository reviewRepository;

    @Autowired
    private ReviewsMadeByUserRepository reviewsMadeByUserRepository;

    @Autowired
    private ReviewsMadeOnGameRepository reviewsMadeOnGameRepository;

    @Autowired
    private UsersLikedRepository usersLikedRepository;

    public List<FullReviewInterfaceProjection> getAllReviews() { return reviewRepository.findAllProjectedBy(); }

    public Optional<FullReviewInterfaceProjection> getReviewById(String id) { return reviewRepository.findById(id); }

    public Optional<GetReviewsMadeFromUserInterfaceProjection> getReviewsMadeByUser(String userId){
        return reviewsMadeByUserRepository.findById(userId);
    }

    public Optional<GetReviewsMadeOnGameInterfaceProjection> getReviewsMadeOnGame(String gameId){
        return reviewsMadeOnGameRepository.findById(gameId);
    }
}
```

Se poate observa că implementarea propriu-zisă este aproape trivială în acest caz. Totuși, o implementare netrivială se poate regăsi în capitolul 6 **Algoritmul de recomandare**.

5.5 Modulul de autentificare al utilizatorilor

Acest modul are un scop relativ simplu, fiind reprezentat de valorificarea fișierului csv în care sunt reținute parolele convertite de funcția hash, dar și de a persista starea de logare a utilizatorului pe partea de front-end a aplicației, prin intermediul unui JWT token. [24]

Implementarea acestuia reprezintă doar un punct de pornire, căci poate fi extins și prin

introducerea autorizării, în care anumite *request-uri* din API pot fi autorizate în funcție de rolul pe care îl are utilizatorul prin intermediul token-ului menționat anterior. Totuși, pentru scopul în mare al proiectului, acela de a observa cum se poate rezolva problema recomandării unor jocuri video, acest aspect de autentificare/autorizare nu a fost atins foarte în detaliu.

Pentru a implementa acest modul, am utilizat funcționalitățile oferite de *Spring Security*. [25]

În general, o aplicație web Spring are la bază un *Dispatcher Servlet*, care, în mod intuitiv conform numelui, acționează ca un dispecer care verifică ce *request* s-a făcut și îl asociază cu unul definit în cadrul unui metode din *Controller*. [26]

Totuși, nu am menționat niciun aspect legat de securitate, iar de acest lucru se ocupă *Spring Security*, care în linii mari include posibilitatea de a adăuga unul sau mai multe filtre înainte de a acționa acel *Dispatcher Servlet*.

Pentru a stabili aceste filtre, am suprascris anumite metode din cadrul clasei abstracte *WebSecurityConfigurerAdapter* în clasa numită *WebSecurityConfig* în felul următor:

Figure 23

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserService userService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.authorizeRequests()
            .anyRequest().permitAll();
        http.addFilter(new CustomAuthenticationFilter(authenticationManagerBean()));
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService);
    }
}
```

În stadiul actual, practic orice *request* are cale liberă spre a fi folosit, însă acest lucru poate fi bineînțeles schimbat ulterior prin autorizarea *request-urilor* în funcție de rolurile utilizatorului.

De asemenea, a fost implementat un *userService* care are rolul de a prelua un utilizator după *username*-ul său.

Punctul de interes îl reprezintă acel *CustomAuthenticationFilter* care prelucrează credențialele (numele de utilizator și parola) din cadrul *request*-ului care se ocupă de autentificarea unui utilizator, și acționează corespunzător dacă autentificarea s-a efectuat cu succes sau nu.

În caz pozitiv, acest *request* va furniza un JWT token ce va fi folosit de către aplicația de front-end pentru a persista logarea utilizatorului.

O variantă de implementare concretă a furnizării aceluia *token* poate fi următoarea:

Figure 24

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
    UserWithUuid user = (UserWithUuid)authResult.getPrincipal();
    Algorithm algorithm = Algorithm.HMAC256("hardcoded-secret".getBytes());
    String accessToken = JWT.create()
        .withSubject(user.getUsername())
        .withExpiresAt(new Date(System.currentTimeMillis() + 60 * 60 * 1000)) //1 hour
        .withIssuer(request.getRequestURL().toString())
        .sign(algorithm);
    response.setHeader("accessToken", accessToken);
    var gson = new Gson();
    Map<String, String> token = new HashMap<>();
    token.put("accessToken", accessToken);
    token.put("uuid", user.getUuid());
    response.setContentType(APPLICATION_JSON_VALUE);
    var writer : PrintWriter = response.getWriter();
    writer.print(gson.toJson(token));
    writer.flush();
}
```

De menționat faptul că acel secret care este pus manual pentru a fi folosit în semnarea *token*-ului prin intermediul algoritmului HMAC nu este o metodă bună în practică (în contextul unei eventuale lansări ipotetice a aplicației în producție), însă în scopul de față, acest aspect nu reprezintă un inconvenient major.

6 Algoritmul de recomandare

Documentația pe care am consultat-o în scopul implementării acestui algoritm constă într-un curs de pe Udemy. [27] Chiar dacă în acest curs noțiunile sunt prezentate având ca limbaj suport Python, intenția mea a fost de a înțelege mai degrabă conceptele și de a le implementa utilizând limbajul Java și baza de date Neo4j.

6.1 Introducere

7 Front-end

8 Manual de utilizare

9 Concluzii și direcții viitoare

Bibliografie

- [1] *IGDB API*
<https://www.igdb.com/api>
<https://github.com/husnjak/IGDB-API-JVM>
- [2] *Steamworks Documentation - Reviews*
<https://partner.steamgames.com/doc/store/getreviews>
- [3] *Java Faker*
<https://github.com/DiUS/java-faker>
- [4] *Apache Commons CSV*
<https://commons.apache.org/proper/commons-csv/>
- [5] *Recommendation systems: Principles, methods and evaluation*, 2015.
F.O. Isinkaye, Y.O. Folajimi, B.A. Ojokoh
<https://www.sciencedirect.com/science/article/pii/S1110866515000341>
- [6] *What is a graph database?*
<https://neo4j.com/developer/graph-database/>
- [7] *Baze de date*
<https://profs.info.uaic.ro/~bd/>
- [8] *Neo4j Training - Querying with Cypher In Neo4j 4.x*
<https://neo4j.com/graphacademy/training-querying-40/01-querying40-introduction-to-cypher/>
- [9] *ASCII art*
https://en.wikipedia.org/wiki/ASCII_art
- [10] *Neo4j Training - Overview of Neo4j 4.x - Neo4j is a Graph Database*
<https://neo4j.com/graphacademy/training-overview-40/01-overview40-neo4j-graph-database/>

- [11] *Graph Modeling Guidelines*
<https://neo4j.com/developer/guide-data-modeling/>
- [12] *Constraints*
<https://neo4j.com/docs/cypher-manual/current/constraints/>
- [13] *APOC Documentation / Introduction*
<https://neo4j.com/labs/apoc/4.1/introduction/>
- [14] *Triggers*
<https://neo4j.com/labs/apoc/4.1/background-operations/triggers/>
- [15] *Hashing a Password in Java*
<https://www.baeldung.com/java-password-hashing>
- [16] *Fisher–Yates shuffle*
https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle
- [17] *Steam Game Review Dataset*
<https://www.kaggle.com/datasets/arashnic/game-review-dataset>
- [18] *Importing CSV Data into Neo4j*
<https://neo4j.com/developer/guide-import-csv/>
- [19] *Large Delete Transaction Best Practices in Neo4j*
<https://neo4j.com/developer/kb/large-delete-transaction-best-practices-in-neo4j/>
- [20] *Drivers & Language Guides*
<https://neo4j.com/developer/language-guides/>
- [21] *Spring Data Neo4j*
<https://neo4j.com/developer/spring-data-neo4j/>
- [22] *Working with Spring Data Repositories*
<https://docs.spring.io/spring-data/neo4j/docs/current/reference/html/#repositories>
- [23] *Spring Dependency Injection*
<https://www.baeldung.com/spring-dependency-injection>
- [24] *Introduction to JSON Web Tokens*
<https://jwt.io/introduction>

[25] *Spring Security*

<https://spring.io/projects/spring-security>

[26] *Spring Security: Authentication and Authorization In-Depth*

<https://www.marcobehler.com/guides/spring-security>

[27] *Recommender Systems and Deep Learning in Python*

<https://www.udemy.com/course/recommender-systems/>