Gestionarea unei biblioteci

Adrian Manea, 510

23 ianuarie 2020

Cuprins

SECȚIUNEA 1

DESCRIEREA APLICAȚIEI

Aplicația servește la gestionarea unei biblioteci, în relația cu cititorii, oferind roluri speciale pentru clienți din mediul academic.

Un cititor poate să se înregistreze cu abonament de tip Edu, care îi oferă accesul la reviste de specialitate, pe lîngă catalogul obișnuit de cărți.

Cititorii care nu au un abonament de tip Edu (NEdu) pot doar să împrumute cărți.

De asemenea, fiecare bibliotecar are cîte o specializare, astfel că cititorii pot să apeleze la bibliotecarul care se potrivește cel mai bine intereselor lor.

Cititorii cu abonament se pot înscrie și pe lista de newsletter, unde pot afla atunci cînd se actualizează stocul de cărți.

După mai multe împrumuturi, un cititor poate să lase feedback pentru un bibliotecar care corespunde specializării pe care el a folosit-o cel mai des.

Pe lîngă cititorii cu abonament activ, restul persoanelor interesate de serviciile bibliotecii pot să consulte catalogul de cărți și își pot crea o listă de dorințe (wishlist) pentru cărțile pe care le-ar dori împrumutate.

1.1 Utilizatori și atribute

Utilizatorii aplicației, identificați prin IP-uri specifice sînt:

- **Administratori** ai bazei de date;
- **Personal educațional** (`EDU`);
- **Cititori** din afara mediului educațional (`NEDU`);
- **Bibliotecari**;
- **Cititori înregistrați, dar neabonați** (`RESTUL`).

Considerăm că studenții și profesorii fac parte din personalul educațional (Edu), statut pe care trebuie să-l verifice periodic. De asemenea, un cititor oarecare poate deveni Edu în timp, dacă adaugă verificarea.

1.2 Procesele din aplicație

(P1) Vizualizarea cărților în stoc;

(P2) Vizualizarea bibliotecarilor, cu specializările lor;

(P3) Adăugarea unei cărți;

(P4) Adăugarea unui cititor;

(P5) Adăugarea unui abonat Edu;

(P6) Înregistrarea unui cont public;

(P7) Actualizarea stocului unor cărți;

(P8) Verificarea statutului Edu;

(P9) Administrarea abonamentului;

(P10) Vizualizarea cărților de o anumită specializare;

(P11) Vizualizarea revistelor de o anumită specializare;

(P12) Adăugare reviste;

(P13) Administrare abonament revistă;

(P14) Administrare abonament la newsletter;

(P15) Împrumut carte;

(P16) Feedback bibliotecar.

De exemplu, **descompunerea funcțională** a procesului (P14), de administrare a abonamentului la o revistă, poate fi reprezentat ca în figura 1.1.

Similar, procesul (P9) de verificare a statutului Edu se poate reprezenta ca în figura 1.2.

Și un ultim exemplu pe care îl prezentăm este acela al procesului (P16), de scriere a feedback-ului pentru un bibliotecar. Descompunerea funcțională este prezentată în figura 1.3.

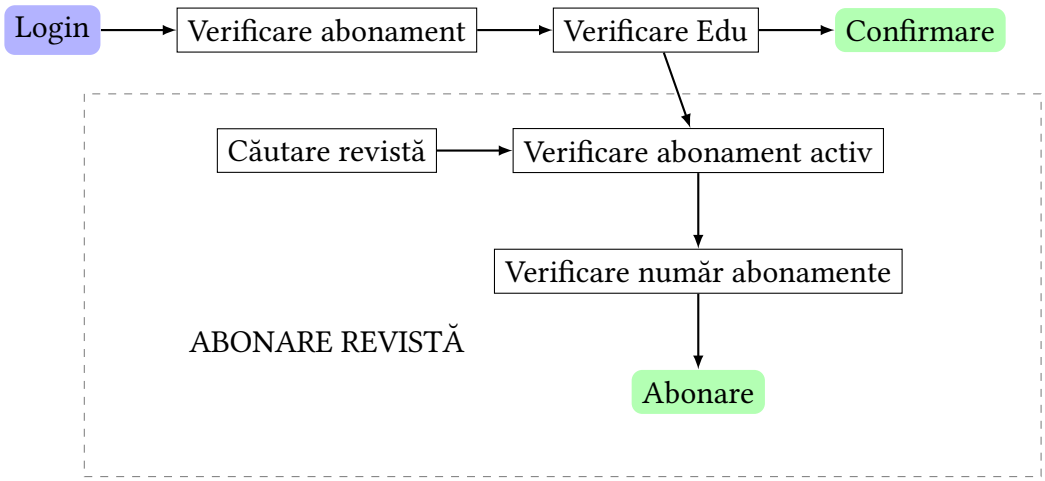

Figura 1.1: Descompunerea funcțională a procesului de abonare la revistă

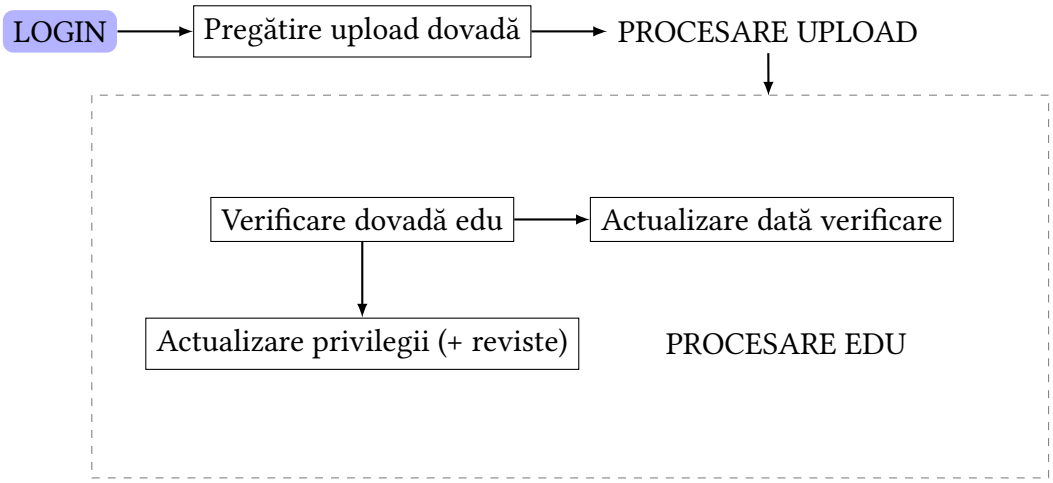

Figura 1.2: Descompunerea funcțională a procesului de verificare Edu

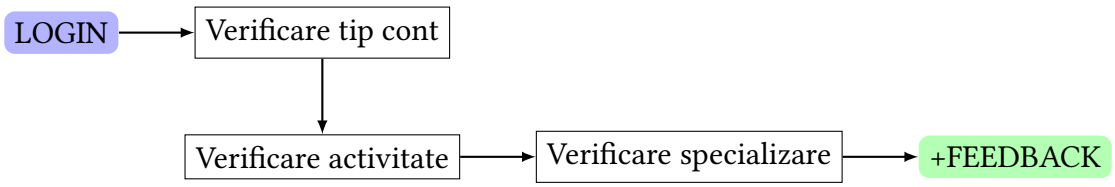

Figura 1.3: Descompunerea funcțională a procesului de adăugare feedback

1.3 Matricea proces-utilizator

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16
Administrator	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Edu	X	X						X	X	X	X			X	X	X
NEdu	X	X						X	X	X				X	X	X
Bibliotecar	X	X	X	X	X				X	X	X	X	X	X		
Public	X									X				X		

1.4 Entitățile și modelarea datelor

Biblioteca este un centru de împrumut care permite și abonamentul la reviste pentru cei din mediul academic.

În bibliotecă se pot adăuga *cărți* și *reviste* de diverse specializări, în funcție de contractele cu furnizorii. Fiecare carte și revistă aparțin unei singure specializări.

Personalul Edu se poate abona și la reviste de specialitate, în baza unei verificări actualizate. Ceilalți cititori cu abonament pot deveni Edu printr-o verificare. Publicul larg poate doar să vadă stocul de cărți, general sau pe specializări și să adauge cărțile dorite în wishlist. Toți utilizatorii se pot abona la newsletter pentru a afla cînd se actualizează stocul.

De asemenea, bibliotecarii sînt asociați specializărilor, fiind responsabil de publicațiile dintr-o anumită specializare.

Cititorii abonați (`EDU` sau `NEDU`) pot lăsa feedback bibliotecarilor cu care au lucrat anterior într-o anumită specializare.

Diagrama ER se poate prezenta ca în figura 1.4.

De menționat că toate relațiile M:M ($\infty : \infty$) vor fi rezolvate cu *tabelele asociative* `BIB_SPEC`, `BIB_CIT` și `PUB_CARTE`, listate la §1.5.

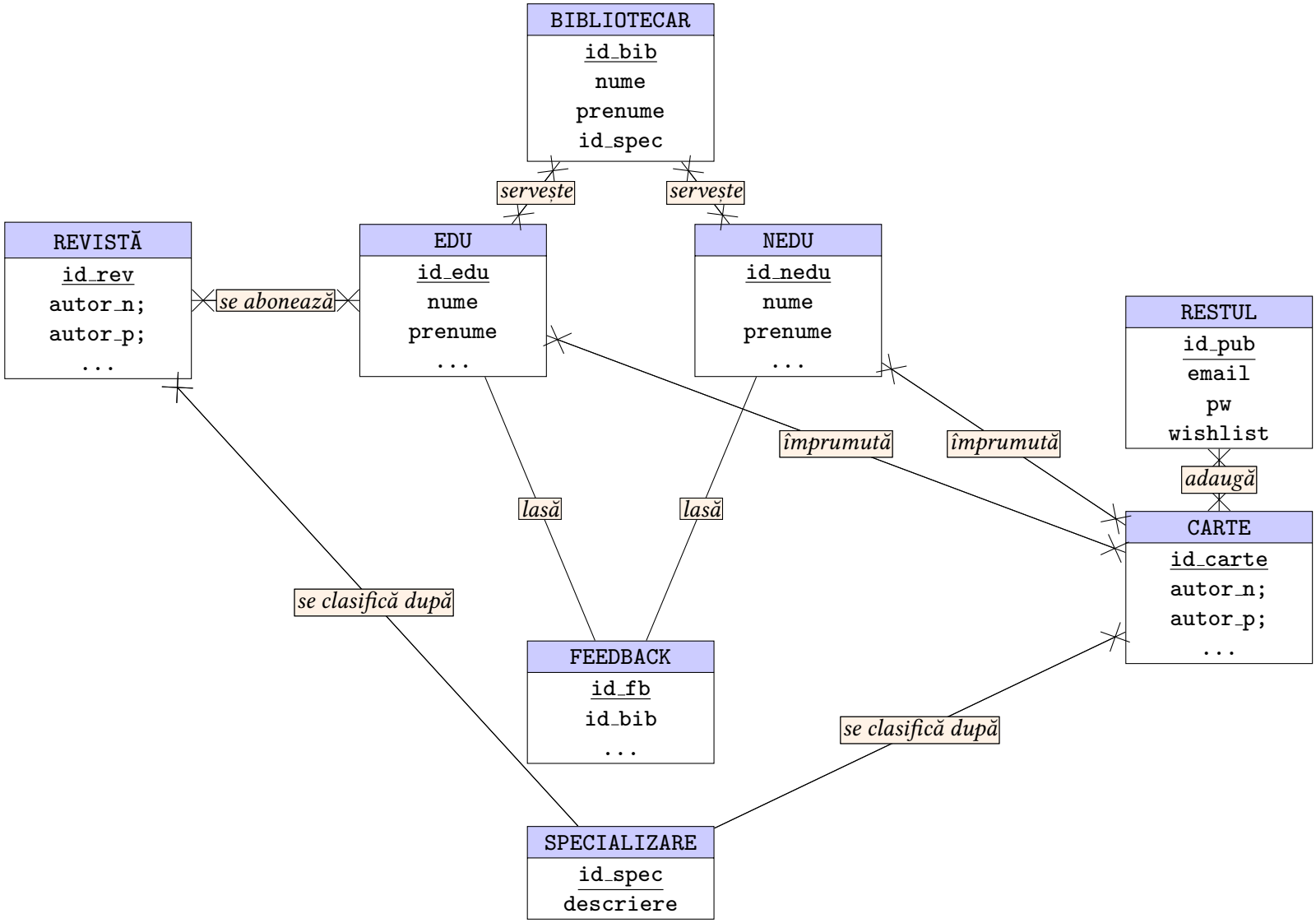

Figura 1.4: Diagrama ER

1.5 Schemele relaționale

```
BIBLIOTECAR     (id_bib#, nume, prenume, id_spec);
NEDU            (id_cit#, nume, prenume, email, newsletter, abonament_activ,
                 id_carte, id_spec);
EDU             (id_edu#, nume, prenume, email, newsletter, abonament_activ,
                 id_carte, id_rev, id_spec);
RESTUL          (id_pub#, email, pw, wishlist, newsletter);
CARTE           (id_carte#, autor_n, autor_p, titlu, an, id_spec, stoc);
REVISTA         (id_rev#, autor_n, autor_p, titlu, numar, id_spec, stoc);
SPECIALIZARE    (id_spec#, descriere);
FEEDBACK        (id_fb#, id_bib, id_cit, id_edu, id_spec, rating,
                 continut, datafb);
BIB_SPEC        (id_bib#, id_spec#);
BIB_CIT         (id_bib#, id_cit#);
PUB_CARTE       (id_pub#, id_carte#);
```

1.6 Matricea entitate-proces

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16
BIBLIOTECAR	S	S	IUD	IUD	IUD		U	IUD	U	S	S	IUD	IUD	U	U	
NEDU	S	S							IUD	IUD	S				U	
EDU	S	S							U	IUD	S	S		U	U	
RESTUL	S					IUD						S		IUD		
CARTE	S		IUD								S				U	
REVISTA												S	I			
SPECIALIZARE											S	S				
FEEDBACK																I

Legenda: I = Insert, U = update, D = delete, S = select.

1.7 Matricea entitate-utilizator

	Edu	NEdu	Bibliotecar	Admin	Restul
BIBLIOTECAR	S	S	S	S, I, U, D	
NEDU		S, I, U, D	S, I, U, D		
EDU		S, I, U, D	S, I, U, D		
RESTUL					
CARTE	S	S	S, I, U, D	S, I, U, D	S
REVISTA	S		S, I, U, D	S, I, U, D	S
SPECIALIZARE	S	S	S	S, I, U, D	S
FEEDBACK	I	I	S	S,I,U,D	

Legenda: I = Insert, U = update, D = delete, S = select.

1.8 Utilizatori și conturi

Conturile utilizatorilor vor fi definite individual pentru utilizatorii identificați de bibliotecă.

În această etapă a dezvoltării aplicației, vor fi disponibile maximum 10000 de conturi Edu, 10000 de conturi de cititori (înregistrați), 10 conturi de bibliotecar, 1000 conturi de utilizator neabonat (RESTUL).

SECȚIUNEA 2

IMPLEMENTARE ÎN POSTGRESQL

2.1 Instalare și configurare inițială

Pentru implementare, vom folosi sistemul de baze de date PostgreSQL, disponibil gratuit și open source.

Implementarea se va face pe laptopul personal, folosind:

- OS: Manjaro Linux i3;
- Emacs 26 pentru editare text și comenzi shell;
- `st` pentru comenzi avansate de terminal, dacă este necesar.

Programul se instalează folosind managerul de pachete din Manjaro, cu comanda:

```
$ sudo pacman -S postgresql
```

După instalare, putem consulta consulta detaliile de pe pagina ArchWiki.

Pe scurt, PostgreSQL creează automat un utilizator cu numele `postgres`, care este proprietarul implicit al bazelor de date. Astfel, pentru a lansa aplicația și a face modificări, trebuie să ne identificăm ca utilizatorul `postgres`, cu una dintre comenzile:

```
$ sudo -iu postgres
$ su -l postgres
```

Inițializarea cluster-ului de baze de date, unde se vor crea bazele și tabelele se face în calitatea de utilizator `postgres`, de exemplu, în locația implicită, cu comanda:

```
[postgres]$ initdb -D /var/lib/postgres/data

# pentru a forța utilizarea limbii engleze și a codării UTF-8, folosim comanda
[postgres]$ initdb --locale=en_US.UTF-8 -E UTF8 -D /var/lib/postgres/data

# ... output ...
# finalizat cu ... ok
```

Folosind `systemd`, trebuie să activăm și să pornim daemon-ul `postgresql`, cu comenzile:

```
$ sudo systemctl enable postgresql
$ sudo systemctl start postgresql
```

Pentru început, creăm un utilizator nou, căruia îi putem da ce atribuții dorim, precum și o bază de date pe care utilizatorul respectiv să o poată accesa (sau administra, în funcție de rolul dat):

```
# devenim utilizatorul postgres mai întîi
$ sudo -iu postgres

# creăm utilizatorul (e.g. theUser) cu dialog pas cu pas, pentru a alege rolul
[postgres]$ createuser --interactive

# creăm o bază de date pentru el (e.g. theDatabase)
[postgres]$ createdb -O theUser theDatabase

# dacă theUser nu are rol de creare, putem crea cu postgres pentru el
[postgres]$ createdb -U postgres -O theUser theDatabase
```

Acum putem porni subshell-ul `psql` și să ne conectăm la baza de date creată mai sus:

```
$ sudo -iu postgres
[postgres]$ psql -d theDatabase

# cîteva dintre meta-comenzile pentru subshell-ul psql:
=> \help                    # accesează help
=> \c <database>            # conectează-te la baza de date <database>
=> \du                      # afișează utilizatorii și permisiunile
=> \dt                      # afișează tabelele și permisiunile
=> \q                       # închide subshell-ul
=> \?                       # afișează toate meta-comenzile
```

2.2 Crearea tabelelor

În primă fază, devenim utilizatorul `postgres` și accesăm baza de date creată:

```
[postgres]$ psql biblioteca
```

Putem încărca un script `psql` din fișierul `tabele.sql`, de exemplu după ce ne-am asigurat că fișierul `tabele.sql` are drepturi corespunzătoare, cu meta-comanda:

```
biblioteca# \i /calea/catre/script/tabele.sql
```

Apoi creăm tabelele corespunzătoare schemei din §1.5 folosind scriptul:

```
create table bibliotecar (
   id_bib bigserial not null primary key,
   -- bigserial = crește singur
   nume varchar(20) not null,
   prenume varchar(20) not null,
   id_spec varchar(20) not null
)

create table edu (
   id_edu bigserial not null primary key,
   nume varchar(20) not null,
   prenume varchar(20) not null,
   email varchar(20),
   newsletter boolean,
   ab_activ boolean not null,
   foreign key (id_carte_edu) references carte(id_carte),
   foreign key (id_rev_edu) references revista(id_rev),
   foreign key (id_spec_edu) references specializare(id_spec)
)

create table nedu (
   id_cit bigserial not null primary key,
   nume varchar(20) not null,
   prenume varchar(20) not null,
   email varchar(20),
   newsletter boolean,
   ab_activ boolean not null,
   foreign key (id_carte_nedu) references carte(id_carte),
   foreign key (id_spec_nedu) references specializare(id_spec)
)

create table restul (
   id_pub bigserial not null primary key,
```

```
   email varchar(20),
   pw varchar(20),
   wishlist text[],-- vector de cuvinte
   newsletter boolean
)

create table carte (
   id_carte bigserial not null primary key,
   autor_n varchar(20) not null,
   autor_p varchar(20) not null,
   titlu varchar(30) not null,
   an smallserial,-- 1 -> 32767
   stoc smallint,-- -32768 ->
   foreign key (id_spec_carte) references specializare(id_spec)
)

create table revista (
   id_rev bigserial not null primary key,
   autor_n varchar(20) not null,
   autor_p varchar(20) not null,
   titlu varchar(30) not null,
   numar smallserial,
   stoc smallint,
   foreign key (id_spec_rev) references specializare(id_spec)
)

create table specializare (
   id_spec bigserial not null primary key,
   descriere text
)

create table feedback (
   id_fb bigserial not null primary key,
   rating smallserial,
   continut text,
   datafb date,
   foreign key (id_bib_fb) references bibliotecar(id_bib),
   foreign key (id_cit_fb) references nedu(id_cit),
   foreign key (id_edu_fb) references edu(id_edu),
   foreign key (id_spec_fb) references specializare(id_spec),
```

```
)

create table bib_spec (
   foreign key (id_bib_bs) references bibliotecar(id_bib) on delete restrict,
   -- nu se șterg tabelele cu referințe pînă ce toate referințele s-au șters
   foreign key (id_spec_bs) references specializare(id_spec) on delete restrict,
)

create table bib_cit (
   foreign key (id_bib_bc) references bibliotecar(id_bib),
   foreign key (id_cit_bc) references nedu(id_cit)
)

create table pub_carte (
   foreign key (id_pub_pc) references restul(id_pub),
   foreign key (id_carte_pc) references carte(id_carte)
)
```

Am adăugat deja constrîngeri în scriptul de mai sus, dar cheile străine pot fi definite și separat, ca în exemplul de mai jos:

```
-- creăm coloana care va deveni cheie străină:
biblioteca# alter table bib_cit
biblioteca# add column id_bib_bc bigint;
ALTER TABLE
biblioteca# alter table bib_cit
biblioteca# add constraint fk_id_bib_bc foreign key (id_bib_bc)
biblioteca# references bibliotecar(id_bib);
ALTER TABLE
```

În felul acesta, legăm coloana `ib_bib_bc` de coloana `bibliotecar(id_bib)`, iar constrîngerea o numim `fk_id_bib_bc`.

După aceea, populăm fiecare dintre tabele cu înregistrări. De exemplu, putem începe cu tabelul care nu conține referințe, `specializare`:

```
biblioteca# insert into specializare values (1, 'matematica'),
                                            (DEFAULT, 'literatura'),
                                            (DEFAULT, 'politica');
INSERT 0 1
```

Valoarea `DEFAULT` continuă automat numărătoarea, deoarece cîmpul respectiv este `id_spec`, declarat cu tipul `bigserial`.

SECȚIUNEA 3

ASPECTE DE SECURITATE

Observație: Pentru simplitate, în cele ce urmează, vom omite prompt-ul de la subshell-ul `psql`, precum și indicațiile de autentificare ca utilizatorul `postgres`. Așadar, vom presupune că utilizatorul s-a identificat deja și s-a conectat la baza de date `biblioteca`, conform indicațiilor din capitolul anterior, pe care le reluăm pentru ultima dată:

```
$ sudo -iu postgres
[postgres@~/]$ psql -d biblioteca
biblioteca# -- comenzile se vor face de la acest nivel
```

Astfel, vom scrie doar `#` ca prefix pentru comenzile introduse de programator, iar răspunsul programului se va nota fără prefix, adică, în general:

```
# comanda psql
# continuare comanda;
RĂSPUNS
continuare răspuns
```

3.1 Criptarea parolelor cu `pgcrypto`

Utilizatorii care au abonament (de tip `EDU` și `NEDU`) își administrează datele și împrumuturile prin interacțiune directă cu bibliotecarii. Dacă vor să lase un feedback, se adresează bibliotecarului, care verifică situația și dă un token unic cu care se autentifică pentru a lăsa feedback.

În schimb, utilizatorii externi (`RESTUL`) își pot crea un cont și se identifică cu email-ul și parola pentru a consulta baza de cărți și stocul. Această parolă va fi stocată în baza de date prin calculul hash-ului, folosind extensia `pgcrypto`. De exemplu, pentru a crea un hash salted al parolei `mypass`, folosind algoritmul blowfish cu 4 runde de criptare, se rulează comenzile (în subshell-ul `psql`):

```
# create extension pgcrypto;
# select crypt('mypass', gen_salt('bf', 4));
CRYPT
------------------------------------------------------------
 $2a$04$1bfMQDOR6aLyD4q3KVb8/ujG7ZAkyie4d/s3ABwuZNbzkFFgXtC76
```

Acum putem introduce în baza de date acest hash pentru cîmpul de parolă. Mai întîi, adăugăm coloana corespunzătoare parolei criptate:

```
# alter table restul
# add column pw_enc varchar(100);
ALTER TABLE
```

Apoi, introducem parolele criptate pentru utilizatorii respectivi:

```
# update restul
# set pw_enc = '$2a$04$1bfMQDOR6aLyD4q3KVb8/ujG7ZAkyie4d/s3ABwuZNbzkFFgXtC76'
# where id_pub = 3;
```

Cum criptarea se face cu o cheie simetrică, putem verifica apoi:

```
# select * from restul where id_pub = 5
# and pw_enc = crypt('mypass', pw_enc);

id_pub  | pw_enc
--------+------------------------------------------------------------
5       | $2a$04$1bfMQDOR6aLyD4q3KVb8/ujG7ZAkyie4d/s3ABwuZNbzkFFgXtC76
(1 row)
```

3.2 Trigger pentru actualizare

Presupunem că bibliotecarul vrea să urmărească actualizarea stocului unor cărți și, de asemenea, că administratorul aplicației vrea să facă o statistică despre utilizatorii neabonați care urmăresc biblioteca (pentru o perioadă fixată de timp, e.g. o lună).

Pentru aceste două scopuri, vom crea cîte un trigger. Primul va urmări cînd se modifică înregistrările din coloana `stoc` a tabelei `carte`, iar al doilea va urmări cînd apar înregistrări noi în tabela `restul`.

Înregistrăm diferențele în tabele separate, `stoc_mod` și, respectiv, `restul_mod`, pe care le creăm corespunzător:

```
# create table stoc_mod (
# id_stoc_mod serial primary key,
# carte_id int not null,
# autor_n varchar(20) not null,
```

```
# autor_p varchar(20) not null,
# titlu varchar(30) not null,
# schimbare_stoc smallint not null,
# schimbat timestamp(6) not null );
```

Apoi se definește funcția corespunzătoare, care urmărește și înregistrează schimbările:

```
# create or replace function log_stoc()
# returns trigger as
# $body$
# begin
# if new.stoc <> old.stoc then
# insert into stoc_mod(id_stoc_mod, carte_id, autor_n,
#                      autor_p, titlu, schimbare_stoc, schimbat)
# values(DEFAULT, old.id_carte, old.autor_n, old.autor_p, old.titlu,
#        new.stoc - old.stoc, now());
# end if;
# return new;
# end;
# $body$ language plpgsql;
CREATE FUNCTION
```

Apoi legăm această funcție de un trigger, care să urmărească tabelul și coloana de `stoc`:

```
# create trigger stoc_modificare
# before update
# on carte
# for each row
# execute procedure log_stoc();
CREATE TRIGGER
```

Modificările pot fi văzute atunci cînd se schimbă o valoare de `stoc` din tabela carte, lucru care poate fi testat astfel:

```
# update carte
# set stoc=5 where id_carte=7;
UPDATE 1
# select * from stoc_mod;
 id_stoc_mod | carte_id | autor_n | autor_p |  titlu  | schimbare_stoc | schimbat
-------------+----------+---------+---------+---------+----------------+-----------------
           1 |        7 | Allah   | Akbar   | Coranul |              9 | 2020-01-18 10:49
           2 |        7 | Allah   | Akbar   | Coranul |             -8 | 2020-01-18 10:52
           3 |        7 | Allah   | Akbar   | Coranul |              4 | 2020-01-18 10:54
(3 rows)
```

Similar, pentru cealaltă situație, creăm un tabel care adaugă noii utilizatori neabonați:

```
# create table restul_mod (
# id_restul_mod serial primary key,
# email_nab varchar(30) not null,
# adaugat timestamp(6) not null );
```

Apoi funcția care urmărește schimbările:

```
# create or replace function nab_nou()
# returns trigger as
# $body$
# begin
# insert into restul_mod(id_restul_mod, email_nab, adaugat)
# values (DEFAULT, new.email, now());
# end if;
# return new;
# end;
# $body$ language plpgsql;
CREATE FUNCTION
```

În fine, trigger-ul, care de data aceasta se execută atunci cînd se adaugă un rînd nou în tabel:

```
# create trigger nab_modificare
# after insert                  -- declanșează doar după INSERT
# on restul
# for each row
# execute procedure nab_nou();
CREATE TRIGGER
```

Și testăm cu:

```
# insert into restul(id_pub, email, pw, newsletter)
# values(DEFAULT, 'newone@y.c', 'IamNewHere', true);
INSERT 1
# select * from restul_mod;

 id_restul_mod | email_nab  |          adaugat
---------------+------------+----------------------------
             3 | newone@y.c | 2020-01-18 11:23:41.204905
(1 row)
```

3.3 Utilizatori și roluri

Utilizatorii pe care îi creăm pot avea mai multe tipuri de privilegii. În ordinea descrescătoare a permisiunilor, acestea sînt:

- SUPERUSER: un astfel de rol are drepturi depline și trece peste orice verificare de securitate. Mai mult decît atît, un utilizator care are acest rol are inclusiv posibilitatea de a mai crea încă unul cu aceleași drepturi. De aceea, rolul SUPERUSER trebuie folosit cu prudență;
- CREATEDB: rolul permite creare bazelor de date;
- CREATEROLE: putem crea încă un alt rol, mai puțin cel de tip SUPERUSER;
- LOGIN: permite utilizatorului să se logheze drept client într-o conexiune cu baza de date.

Sintaxa pentru crearea rolurilor este simplă:

```
# -- crearea rolului
# create role unRol;
CREATE ROLE
# -- ștergerea rolului creat;
# drop role unRol;
DROP ROLE
```

Implicit, dacă nu se specifică exact rolul pe care să îl aibă utilizatorul, el nu va avea niciun drept. Apoi, dacă vrem să creăm utilizatori cu roluri anume, se pot atribui explicit sau se pot modifica rolurile existente:

```
# -- utilizatorul are parolă, dar nu poate face nimic cu ea
# create role nolog_user with password 'pass1';
CREATE ROLE
# -- utilizator cu parolă, cu drepturi de LOGIN
# create role log_user with login password 'pass2';
CREATE ROLE
# -- modificăm rolul și mai adăugăm privilegii
# alter role log_user createrole createdb;
ALTER ROLE
```

Verificarea se poate face în tabela pg_role, cu sau fără filtrare:

```
# select rolcreaterole, rolcreatedb from pg_roles where rolname='log_user';

 rolcreaterole | rolcreatedb
---------------+-------------
 t             | t
(1 row)
```

Alternativ, putem verifica toate rolurile folosind meta-comanda \du, care afișează toate rolurile și dacă sînt incluse în grupuri.

3.3.1 Fișierul de configurare `pg_hba.conf`

Setările privitoare la roluri și modurile de autentificare sînt salvate în fișierul `pg_hba.conf`, pe care îl putem găsi cu o comandă de forma:

```
# select name, setting from pg_settings where name like '%hba';

 name     | setting
----------+-------------------------------------
 hba_file | /etc/postgresql/10/main/pg_hba.conf
(1 row)
```

Acest fișier specifică rolurile, drepturile și modurile de conectare la bazele de date (prin arhitectura client-server sau local). Mai multe detalii pot fi găsite în documentația oficială ([pgoff]). Deoarece vom folosi doar o conexiune locală, în continuare ne vom concentra pe această modalitate. Pentru aceasta, avem sintaxa generală:

```
local      database  user  auth-method  [auth-options]
```

Metodele de autentificare (`auth-method`) sînt multiple, dintre ele menționăm:

- `trust`: conexiunea se face automat, fără verificare;
- `reject`: conexiunea se respinge automat;
- `password`: se cere clientului o parolă, care se verifică în text clar;
- `md5, scram-sha-256`: se cere clientului o parolă, care se verifică prin metodele de hashing din nume.

Pentru cele două roluri din exemplul de mai sus, în fișierul `pg_hba.conf` avem intrările:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             nolog_user                              password
local   all             log_user                                password
```

Putem verifica acest lucru dintr-un rol de `SUPERUSER` cu următorul query:

```
# SELECT database, user_name, auth_method
# FROM pg_hba_file_rules
# WHERE CAST(user_name AS TEXT) LIKE '%log_user%';

 database | user_name    | auth_method
----------+--------------+-------------
    {all} | {nolog_user} | password
    {all} | {log_user}   | password
```

Apoi, deoarece utilizatorul `nolog_user` nu se poate autentifica, primim erori, chiar dacă acestuia i-a fost atribuită o parolă:

```
[postgres]$ psql -U nolog_user -W postgres
Password for user nolog_user:
psql: FATAL: role "nolog_user" is not permitted to log in
[postgres]$ psql -U log_user -W postgres
Password for user log_user:
psql (12.1)
Type "help" for help.
postgres=>
```

3.3.2 Atribuirea (GRANT) permisiunilor

Pentru ca rolurile create să poată folosi obiectele din bazele de date (tabele, vizualizări, coloane, funcții etc.), trebuie să le fie acordate privilegii în acest sens. Acest lucru se face cu comanda `GRANT`.

De exemplu, putem crea un rol care poate doar să vizualizeze unele coloane dintr-un anumit tabel. În cazul nostru, putem permite unui bibliotecar să vadă doar numele, prenumele, email-ul și statutul abonamentului cititorilor `EDU` sau `NEDU`, fără alte informații. De asemenea, bibliotecarul respectiv poate modifica doar cîmpul `ab_activ` pentru fiecare cititor.

Pentru aceasta, creăm utilizatorul corespunzător, cu credențiale și drepturi de login:

```
# create role bib1 with login password 'ilovebooks';
CREATE ROLE
# grant select (nume, prenume, email, ab_activ) on table edu to bib1;
GRANT
# grant select (nume, prenume, email, ab_activ) on table nedu to bib1;
GRANT
# grant update (ab_activ) on table edu to bib1;
GRANT
# grant update (ab_activ) on table nedu to bib1;
GRANT
```

Ne conectăm apoi cu utilizatorul `bib1` la baza de date `biblioteca` și testăm:

```
[postgres]$ psql -U bib1 -W biblioteca
Password:
psql (12.1)
# select * from edu;
ERROR: permission denied for table edu
# select select nume from edu;
    nume
------------
 Ionescu
```

```
 Marcelescu
 Georgescu
(3 rows)
# update edu
# set ab_activ='f' where nume='Ionescu';
UPDATE 1
```

Similar, am mai creat în aplicație două roluri pentru cititori, unul fără cont edu care poate doar să vadă cărțile din stoc și unul cu cont edu, care poate să vadă și revistele. Ambii au drepturi de login, cu parole asociate:

```
# create role cititor with login password 'ilovetoread';
CREATE ROLE
# grant select on table carte to cititor;
GRANT
# create role cititorEdu with login password 'iloveresearch';
CREATE ROLE
# grant select on table carte to cititorEdu;
GRANT
# grant select on table revista to cititorEdu;
GRANT
```

SECȚIUNEA 4

ALTE ASPECTE ȘI IDEI

4.1 Auditarea modificărilor cu `pgaudit`

În continuare, putem să activăm funcția de auditare din extensia `pgaudit`, care va afișa într-un log modificările făcute la baza de date, filtrate după tipul modificării. De exemplu, se pot înregistra în log operații de tipul:

- `READ`, adică atunci cînd se fac operații de citire de tipul `SELECT, COPY` sau un query;
- `WRITE`, cînd se introduc, se modifică sau se șterg valori ori alte operații care schimbă conținutul înregistrărilor;
- `FUNCTION`, atunci cînd se execută funcții sau blocuri `DO`;
- `ROLE`, pentru cînd se atribuie, șterg sau modifică roluri și utilizatori noi;
- `MISC`, pentru alte tipuri de operații (`DISCARD, VACUUM, FETCH` etc.).

Să creăm un tabel care să înregistreze datele pentru audit cînd se citesc valori:

```
# create table pgAuditX( id serial, continut text );
# insert into pgAuditX(id, continut) values(DEFAULT, 'test audit');
```

Acum trebuie să activăm extensia de audit care să urmărească operațiile de tip `READ`:

```
# alter system set pgaudit.log to 'read';
# select pg_reload_conf();
# -- test:
# select continut from pgAuditX;
```

Acum verificăm în terminal, în afara bazei de date:

```
$ grep AUDIT postgresql-Sat.log | grep READ

2020-01-18 postgres postgres LOG:
AUDIT: SESSION,1,1,READ,SELECT,,,SELECT name FROM pgAuditX;,<none>
```

4.2 Mascarea informațiilor cu `postgresql_anonymizer`

Extensia `postgresql_anonymizer` permite mascarea informațiilor sensibile și înlocuirea lor cu unele aleatorii, astfel încît informațiile inițiale să poată fi recuperate. Anonimizarea se face pe baza etichetelor de securitate din PostgreSQL ([psqlsecl]). Un exemplu de bază este:

```
# create extension if not exists anon cascade;
# select anon.load();
# create table player(id serial, name text, points int);
# security label for anon on column player.name
# is 'masked with function anon.fake_last_name()';
# security label for anon on column player.id
# is 'masked with value null';
```

Anonimizarea poate fi făcută `in-place`, fie folosind funcția `anon.anymize_database()`, care este *destructivă* (i.e. înlocuiește datele cu cele anonimizate):

```
# select * from customer;

 id  |   full_name      |   birth    |    employer    | zipcode | fk_shop
-----+------------------+------------+----------------+---------+---------
 911 | Chuck Norris     | 1940-03-10 | Texas Rangers | 75001   | 12
 112 | David Hasselhoff | 1952-07-17 | Baywatch      | 90001   | 423

# create extension if not exists anon cascade;
# select anon.load();
# security label for anon on column customer.full_name
# is 'masked with function anon.fake_first_name() || '' '' || anon.fake_last_name()';
# security label for anon on column customer.birth
# is 'masked with function anon.random_date_between(''01/01/1920''::DATE,now())';
# security label for anon on column customer.employer
# is 'masked with function anon.fake_company()';
# security label for anon on column customer.zipcode
# is 'masked with function anon.random_zip()';
# select anon.anonymize_database();
# select * FROM customer;

 id  |     full_name     |   birth    |     employer     | zipcode | fk_shop
-----+-------------------+------------+------------------+---------+---------
```

```
 911 | michel Duffus     | 1970-03-24 | Body Expressions | 63824   | 12
 112 | andromache Tulip  | 1921-03-24 | Dot Darcy        | 38199   | 423
```

Sau putem doar să ascundem temporar informațiile, declarîndu-le cu eticheta `MASKED`. Informațiile inițiale sînt:

```
# select * from people;
 id | fistname | lastname |   phone
----+----------+----------+------------
 T1 | Sarah    | Conor    | 0609110911
(1 row)
```

Pentru scopul nostru, mai întîi activăm extensia:

```
# create extension if not exists anon CASCADE;
# select anon.start_dynamic_masking();
```

Declarăm un utilizator ce va fi mascat:

```
# create role skynet login;
# security label for anon on role skynet is 'masked';
```

Apoi regulile de mascare:

```
# security label for anon on column people.lastname
# is 'masked with function anon.fake_last_name()';
# security label for anon on column people.phone
# is 'masked with function anon.partial(phone,2,$$******$$,2)';
```

În fine, ne conectăm cu utilizatorul mascat și găsim informațiile ascunse:

```
# \! psql peopledb -U skynet -c 'select * from people;'
 id | fistname | lastname  |   phone
----+----------+-----------+------------
 T1 | Sarah    | Stranahan | 06******11
(1 row)
```

INDEX

BIBLIOGRAFIE

[archwiki] Comunitatea ArchLinux. Create your first database, 2020. https://wiki.archlinux.org/index.php/PostgreSQL#Create_your_first_database/user.

[juba] Salahaldin Juba and Andrey Volkov. *Learning PostgreSQL 11.* Packt, 2019.

[pganon] Damien Clochard. PostgreSQL anonymizer, 2020. https://labs.dalibo.com/postgresql_anonymizer.

[pgaudit] Jason O'Donnell. pgAudit: Auditing database operations, 2016. https://info.crunchydata.com/blog/pgaudit-auditing-database-operations-part-1.

[pgauditgh] Comunitatea pgAudit. pgaudit readme, 2020. https://github.com/pgaudit/pgaudit/blob/master/README.md.

[pgoff] Comunitatea PSQL. The pg_hba.conf file, 2020. https://www.postgresql.org/docs/current/auth-pg-hba-conf.html.

[psqlcrypto] Fujitsu Enterprise. How to use pgcrypto to further protect your data, 2019. https://www.postgresql.fastware.com/blog/further-protect-your-data-with-pgcrypto.

[psqlsecl] Comunitatea PSQL. Security label, 2020. https://www.postgresql.org/docs/current/sql-security-label.html.

[psqltut] PostgreSQL Tutorial. Creating a Trigger in PostgreSQL, 2020. https://www.postgresqltutorial.com/creating-first-trigger-postgresql/.