Cinema

Seminar 7 – Persistența în baza de date (JDBC, PostgreSQL)

Enunț

În cursul 6 am implementat `UtilizatorDbRepository`: repository-ul rețelei sociale salvează utilizatorii în tabela `users`, iar `Main` lucrează doar cu interfața `Repository`. În seminar aplicăm același model pe un domeniu nou: filmele unui cinematograf, salvate în baza de date `cinema`. Pornim de la proiectul `curs6_handson`; proiectul seminarului este `sem7-PersistentaDB`.

- Tabela `filme(id, titlu, regizor, an, durata)`; id-ul îl generează baza de date (`GENERATED ALWAYS AS IDENTITY`), deci nu îl dăm la `INSERT`.
- `FilmDbRepository implements Repository<Long, Film>`: fiecare metodă își deschide `Connection`, `PreparedStatement` și `ResultSet` în try-with-resources.
- `SQLException` se transformă în `RepositoryException`; erorile de validare rămân `ValidationException`.
- `RepositoryFactory` creează și repository-ul de filme; `Main` nu cunoaște clasa concretă.

Încălzire. Baza de date cinema în pgAdmin

> Creați în pgAdmin baza de date `cinema` și rulați scriptul `sql/cinema.sql` în Query Tool. Scrieți interogările: (a) filmele apărute după 2010, ordonate după an; (b) numărul de filme și durata medie pentru fiecare regizor; (c) inserarea unui film cu id-ul dat explicit (`id = 100`). Ce mesaj primiți la (c)?

Întrebări

1. Ce înseamnă `GENERATED ALWAYS AS IDENTITY`? De ce eșuează inserarea de la (c)?

   > **Răspuns:** Baza de date generează id-ul dintr-o secvență, la fiecare `INSERT`. Cu `ALWAYS`, refuză orice valoare dată de noi: „cannot insert a non-DEFAULT value into column id”. Cu `GENERATED BY DEFAULT` ar accepta și id-uri date explicit. Avantajul: id-urile sunt unice chiar dacă mai mulți clienți inserează simultan, iar aplicația nu mai calculează „următorul id”.

2. Scrieți interogarea (b).

   ```
   SELECT regizor, COUNT(*) AS nr, ROUND(AVG(durata)) AS durata_medie
   FROM filme
   GROUP BY regizor
   ORDER BY nr DESC;
   ```

   > **Răspuns:** Rezultat: Christopher Nolan 3 filme (166 minute), Quentin Tarantino 2 (160), ceilalți câte unul. `GROUP BY` formează câte un grup pentru fiecare regizor, iar `COUNT` și `AVG` se calculează pe fiecare grup.

3. Scriptul poate fi rulat de mai multe ori fără erori și fără filme dublate. Ce asigură asta?

   > **Răspuns:** `CREATE TABLE IF NOT EXISTS` nu mai creează tabela a doua oară, iar `WHERE NOT EXISTS` inserează un film doar dacă titlul lui nu există. Id-ul generat nu împiedică dublurile: același film inserat de două ori primește două id-uri diferite. Doar o constrângere `UNIQUE` pe titlu le-ar interzice în baza de date.

Soluție (sql/cinema.sql)

Cerința 1. Entitatea Film și validatorul

> În `curs.domain`, creați clasa `Film extends Entity<Long>` cu câmpurile `titlu`, `regizor`, `an` și `durata` (minute), un constructor **fără** id, getteri, setteri și `toString`.

În `curs.domain.validators`, creați `FilmValidator implements Validator<Film>`: titlul și regizorul nu lipsesc, anul este între 1888 și anul curent, durata este între 1 și 600 de minute; id-ul poate lipsi, dar, dacă există, este pozitiv. Raportați toate erorile într-o singură `ValidationException`. În `Main.validare`, validați Barbie (Greta Gerwig, 2023, 114) și un film cu titlul gol, anul 1800 și durata 0.

(axa de schimbare: o entitate nouă în aceeași arhitectură)

Întrebări

1. De ce constructorul lui `Film` nu primește id-ul, iar validatorul acceptă un id `null`?

   Răspuns: Un film nou nu are încă identitate: id-ul apare abia după `INSERT`, generat de baza de date, și îl setează repository-ul. Validatorul verifică datele introduse de utilizator. La `update`, id-ul este necesar, pentru că spune ce rând se modifică.

2. De ce raportăm toate erorile odată, în loc să aruncăm excepția la prima eroare?

   Răspuns: Utilizatorul vede dintr-o dată tot ce trebuie corectat: „titlul lipseste; an invalid (1888 - 2026); durata invalida (1 - 600 minute)". Cu excepție la prima eroare, ar corecta câte un câmp și ar încerca din nou de trei ori.

3. Limita superioară a anului este `Year.now().getValue()`, nu valoarea 2026 scrisă în cod. De ce?

   Răspuns: Regula rămâne corectă și în anii următori, fără să modificăm codul. O constantă ar respinge, din ianuarie, filmele anului nou.

Soluție (Film, FilmValidator, Main.validare)

Cerința 2. Citirea filmelor: findAll și findOne

În `curs.repository.db`, creați `FilmDbRepository implements Repository<Long, Film>`, cu constructorul `(validator, url, username, password)`. Implementați `findAll` (toate filmele, ordonate după id) și `findOne(id)` (`WHERE id = ?`; `Optional.empty()` dacă filmul nu există; `IllegalArgumentException` pentru id `null`).
Fiecare metodă deschide `Connection`, `PreparedStatement` și `ResultSet` în try-with-resources și transformă `SQLException` în `RepositoryException`. În `Main.readAll` și `Main.readOne`, afișați: toate filmele; primul film; filmul cu id 9999; filmul cu id `null`.

(axa de schimbare: alt tip de stocare pentru aceeași interfață)

Întrebări

1. În `findOne`, `ResultSet`-ul are propriul try-with-resources, în interiorul primului. De ce nu îl deschidem în același try cu `Connection` și `PreparedStatement`, ca în `findAll`?

   Răspuns: Înainte de `executeQuery` trebuie setat parametrul, cu `statement.setLong(1, id)`. Între resursele aceluiași try nu putem scrie instrucțiuni, deci `ResultSet`-ul se deschide după `setLong`, într-un try separat. În `findAll` nu există parametri.

2. De ce `findOne` folosește `if (resultSet.next())`, iar `findAll` folosește `while`?

   Răspuns: După `executeQuery`, cursorul este **înaintea** primului rând, iar `next()` îl mută pe rândul următor și întoarce `false` dacă nu mai există rânduri. `id` este cheie primară, deci `findOne` găsește cel mult un rând.

3. De ce `findAll` nu declară `throws SQLException`?

> **Răspuns:** Interfața `Repository` nu declară excepții checked, iar o implementare nu poate adăuga excepții checked noi. Chiar dacă ar putea, `SQLException` ar obliga `Main` și toate straturile să știe că lucrăm cu o bază de date. `RepositoryException` este unchecked și păstrează `SQLException` drept cauză.

4. `resultSet.getInt("an")` întoarce 0 pentru o valoare `NULL`. De ce nu ne afectează asta?

> **Răspuns:** Coloanele tabelei sunt `NOT NULL`, deci baza de date nu acceptă rânduri fără an sau durată. Pentru coloanele care pot fi `NULL`, se verifică `resultSet.wasNull()` sau se citește cu `getObject("an", Integer.class)`.

Soluție (FilmDbRepository.findAll, FilmDbRepository.findOne, Main.readAll, Main.readOne)

Cerința 3. Create, update, delete

> Implementați în `FilmDbRepository`: `save` – validare, `INSERT` fără id, apoi citirea id-ului generat cu `getGeneratedKeys()` și setarea lui pe film; `update` – validare, `UPDATE ... WHERE id = ?`, iar dacă nu s-a modificat niciun rând, întoarceți `Optional.of(entity)`; `delete` – întoarce filmul șters sau `Optional.empty()`.
> În `Main.create`, `Main.update` și `Main.delete`, testați toate scenariile: film valid, invalid, `null`, inexistent (id 9999) și, la ștergere, a doua ștergere a aceluiași film. Afișați mesaje specifice, de exemplu „Filmul nu este valid: ...” sau „Film inexistent (id 9999)”.

(axa de schimbare: operații noi pe aceeași tabelă)

Întrebări

1. De ce scriem `prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)`? Ce s-ar întâmpla dacă `save` nu ar seta id-ul pe film?

> **Răspuns:** Cere driverului să întoarcă, după `executeUpdate`, valorile generate de baza de date, citite cu `getGeneratedKeys()`. Fără ele, apelantul nu ar ști id-ul filmului salvat și ar trebui să-l caute după titlu, ceea ce nu e sigur: pot exista două filme cu același titlu.

2. Cum aflăm, la `update`, că filmul nu există, fără un `SELECT` înainte?

> **Răspuns:** `executeUpdate()` întoarce numărul de rânduri modificate. 0 înseamnă că niciun rând nu are id-ul dat.

3. De ce `delete` apelează întâi `findOne`?

> **Răspuns:** Contractul cere să întoarcem filmul șters, iar după `DELETE` rândul nu mai există. În PostgreSQL, același lucru se poate obține într-o singură interogare: `DELETE FROM filme WHERE id = ? RETURNING *`.

4. `Main.update` află id-ul filmului Barbie cu `repo.findAll(f -> f.getTitlu().equals("Barbie")).getLast().getId()`. De ce `getLast`? Ce se întâmplă dacă nu există niciun film Barbie?

> **Răspuns:** Dacă o rulare anterioară s-a oprit înainte de ștergere, în tabelă pot exista mai multe filme Barbie. Lista este ordonată după id, deci `getLast` alege ultimul salvat. Pentru o listă vidă, `getLast` aruncă `NoSuchElementException`.

Soluție (FilmDbRepository.save, update, delete; Main.create, Main.update, Main.delete)

Cerința 4. Factory pentru filme

> Adăugați în `RepositoryFactory` metoda `createFilmRepository(Validator<Film> validator)`, construită ca `createUtilizatorRepository`, cu un `Map<RepositoryType, Supplier<...>>`: `DATABASE` → `FilmDbRepository`, `MEMORY` → `InMemoryRepository`. Pentru `FILE`, aruncați `IllegalArgumentException`.
> În `config.properties`, `db.url` indică baza de date `cinema`. În `Main.creeazaRepo`, obțineți repository-ul din factory; în `Main.factoryFile`, cereți un repository de filme de tip `FILE` și afișați mesajul primit.

(axa de schimbare: entități noi)

Întrebări

1. Ce am modificat în `Repository` și `InMemoryRepository` ca să lucrăm cu filme?

   > **Răspuns:** Nimic: ambele sunt generice, `<ID, E extends Entity<ID>>`. Am adăugat doar clase noi și o metodă în factory (OCP). `Main` lucrează cu `Repository<Long, Film>` și nu cunoaște `FilmDbRepository`.

2. Profesorul are serverul pe portul 5436, studenții pe 5432. Unde se face schimbarea? De ce nu în `FilmDbRepository`?

   > **Răspuns:** În `config.properties`, la `db.url`. Repository-ul primește URL-ul prin constructor, de la factory, deci nu depinde de un calculator anume. Configurația se schimbă fără recompilare.

3. Cu `repository.type=MEMORY`, putem salva un film nou?

   > **Răspuns:** Nu. `InMemoryRepository.save` cere id-ul, pentru că nu are cine să-l genereze, iar un film nou nu are id: aruncă `IllegalArgumentException`. Ca să meargă, repository-ul în memorie ar trebui să genereze singur id-urile, de exemplu cu un contor. Generarea id-ului ține de stocare, nu de entitate.

Soluție (RepositoryFactory.createFilmRepository, config.properties, Main.creeazaRepo, Main.factoryFile)

Cerința 5. Căutare în SQL sau în memorie

> Adăugați în `FilmDbRepository` metoda `List<Film> findByRegizor(String regizor)`, cu `WHERE regizor = ? ORDER BY an`. În `Main.cautaRegizor`, afișați filmele lui Christopher Nolan în două feluri: cu `findByRegizor` și cu metoda default `findAll(Predicate)` din `Repository`.

(axa de schimbare: căutări specifice unei stocări)

Întrebări

1. Rezultatul este același. Ce diferă?

   > **Răspuns:** Cu `findByRegizor`, baza de date filtrează și trimite doar rândurile potrivite. Cu `findAll(Predicate)`, se citesc toate filmele, iar filtrarea se face în Java. Pentru tabele mari, varianta cu `WHERE` este mult mai rapidă și poate folosi un index.

2. De ce `findByRegizor` nu este în interfața `Repository`? Cum o apelăm din `Main`, unde `repo` are tipul `Repository<Long, Film>`?

   > **Răspuns:** Este specifică filmelor: în `Repository` ar trebui implementată și pentru utilizatori și în toate stocările (ISP). În `Main` folosim `if (repo instanceof FilmDbRepository dbRepo)`, dar astfel `Main` cunoaște clasa concretă. Varianta mai bună este o interfață `FilmRepository extends Repository<Long, Film>`, cu metoda `findByRegizor`.

3. Ce se întâmplă dacă scriem `"... WHERE regizor = '" + regizor + "'"`, iar regizorul este `x' OR '1'='1`?

   > **Răspuns:** Textul primit devine cod SQL: condiția devine `regizor = 'x' OR '1'='1'`, mereu adevărată, și primim toate filmele (SQL injection). Cu `?` și `setString`, valoarea este trimisă separat și rămâne text: nu există niciun regizor cu acest nume.

Soluție (FilmDbRepository.findByRegizor, Main.cautaRegizor)

Cerința 6. Rapoarte cu stream-uri

> În `Main.rapoarte`, folosind `repo.stream()`, afișați: (a) numărul de filme pentru fiecare regizor (`groupingBy`, `counting`); (b) durata medie (`mapToInt`, `average`); (c) titlul celui mai lung film (`max`, `Comparator.comparingInt`); (d) titlurile filmelor apărute după 2010, ordonate după an.

(axa de schimbare: rapoarte noi)

Întrebări

1. Raportul (a) poate fi calculat și de baza de date. Cum? Când preferăm fiecare variantă?

   ```
   SELECT regizor, COUNT(*) FROM filme GROUP BY regizor;
   ```

   > **Răspuns:** SQL, când datele sunt multe: se transmite doar rezultatul. Stream-ul, când datele sunt deja în memorie sau prelucrarea se scrie mai ușor în Java. Stream-ul merge la fel pe orice repository, inclusiv pe cel în memorie.

2. De ce `average()` întoarce `OptionalDouble`, iar `max(...)` întoarce `Optional<Film>`?

   > **Răspuns:** Pentru o tabelă goală nu există medie și nici cel mai lung film. Tipul de retur ne obligă să tratăm și acest caz, în loc să primim 0 sau `null`.

3. `FilmDbRepository` nu implementează `stream()`. De unde vine? Câte interogări SQL execută `Main.rapoarte`?

   > **Răspuns:** `stream()` este metodă default în `Repository` și este construită din `findAll()`. Fiecare apel `repo.stream()` execută un nou `SELECT`, deci `rapoarte` execută patru interogări. Pentru un singur `SELECT`, se citește lista o dată și se construiesc stream-urile din ea.

Soluție (Main.rapoarte)