CZU: 372.8004:004.655.3

ABORDĂRI METODOLOGICE ÎN APLICAREA TEHNICII ORM.

Olga CERBU[1], doctor, conferențiar universitar

Gabriel TUREȚCHI[2], student

olga.cerbu@gmail.com

[1]Universitatea de Stat din Moldova, [2]Universitatea Tehnică a Moldovei

Rezumat. *În lucrare este abordată aplicarea tehnicii ORM, ORM înseamnă maparea obiect-relațională, în care obiectele sunt folosite pentru a conecta limbajul de programare la sistemele de baze de date, cu posibilitatea de a lucra cu SQL și concepte de programare orientată pe obiecte. Se operează asupra obiectelor. Întreaga metodologie urmată de ORM-uri depinde de paradigma orientată pe obiecte. ORM-urile generează obiecte care se mapează la tabelele din baza de date virtual. Odată ce aceste obiecte sunt ridicate, atunci programatorii pot lucra cu ușurință pentru a prelua, manipula sau șterge orice câmp din tabel fără a acorda prea multă atenție limbajului în mod specific. Acceptă scrierea de interogări SQL lungi și complexe într-un mod mai simplu.*

Cuvinte cheie: *ORM, mapare, relație, programarea orientată pe obiecte, SQL.*

Summary. *The paper addresses the usage of ORM, ORM means object-relational mapping, in which objects are used to connect programming language to database systems, with the ability to work with SQL and object-oriented programming concepts. It operates on objects. The whole methodology followed by ORMs depends on the object-oriented paradigm. ORMs generate objects that are mapped to tables in the virtual database. Once these objects are raised, then programmers can easily work to retrieve, manipulate, or delete any field in the table without paying too much attention to the language specifically. It supports writing long and complex SQL queries in a simpler way.*

Keywords: *ORM, mapping, relation, object-oriented programming, SQL.*

Introducere

Maparea obiect-relațională (ORM, O/RM și O/R mapping tool) în informatică este o tehnică de programare pentru conversia datelor între sisteme de tip incompatibil în limbaje de programare orientate pe obiecte. ORM este o tehnică de programare ce face posibilă accesarea și manipularea obiectelor fără ca programatorii să fie interesați de sursa de date de unde provin aceste obiecte. Această tehnică a apărut din nevoia de a depăși diferențele de paradigmă dintre modelul orientat pe obiecte (susținut de limbajele de programare de nivel înalt actuale) și modelul relațional (utilizat de cele mai populare sisteme de gestiune a bazelor de date). Limbajele de programare orientate pe obiecte reprezintă datele într-un graf interconectat de obiecte, pe când bazele de date relaționale folosesc un mod tabelar de reprezentare. Efortul de a conecta atributele claselor definite prin intermediul unui limbaj orientat pe obiecte cu câmpurile tabelelor din baza de date nu poate fi ignorat, iar scopul unui ORM este acela de a crea o relație naturală, transparentă, fiabilă și de durată între cele

două modele. Această nepotrivire de paradigmă pare să nu își fi găsit încă o soluționare definitivă care să fie aprobată de toți programatorii din industria IT, însă opinia generală este aceea că framework-urile ORM reprezintă un important pas înainte.

Procesul automat de stocare a obiectelor într-o bază de date relațională folosind un framework ORM, constă în maparea obiectelor la tabelele corespunzătoare, asocierea dintre ele fiind descrisă folosind metadata. Un framework ORM complet include următoarele funcționalități:

- un API pentru operațiile CRUD (create, retrive, update, delete) aferente claselor persistente;
- un limbaj pentru specificarea interogărilor adresând clasele persistente și atributele acestora;
- un mod care să faciliteze definirea metadata pentru mapările dintre obiect și tabelă;
- o abordare consistentă a tranzacțiilor, a metodelor de stocare a datelor („caching") și a asocierilor dintre clase;
- tehnici de optimizare în funcție de natura aplicației [1].

În acest articol se abordează, în ansamblu, decizia de implementare a framework-urilor ORM prin descrierea celor mai importante provocări și prin argumentarea pro și contra în aceste cazuri. Implementările sunt făcute în limbajul Java și framework-ul ORM Hibernate, care sunt foarte populare în rândul programatorilor.

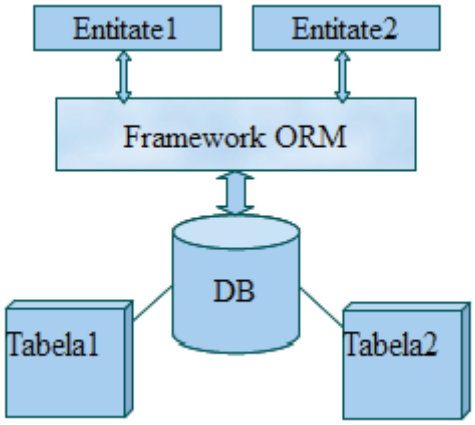

Fig. 1. Componentele implicate în mecanismul ORM. Cele două entități sunt persistate în tabelele corespunzătoare prin intermediul framework-ului ORM

Într-un mediu concurențial partajarea resurselor devine extrem de importantă și problema menținerii integrității datelor tinde sa devină complexă. Din acest motiv modul în care această partajare se implementează trebuie să fie cât mai simplu, să corespundă specificațiilor proiectului și să asigure integritatea datelor. În continuare vor fi descrise câteva dintre tehnicile de manipulare a datelor care trebuie folosite la un moment dat de programatori și care implică o atenție deosebită în ceea ce privește partajarea resurselor.

Blocarea optimistă a resurselor într-un mediu concurențial („optimistic concurrency")

Blocarea resurselor este folosită pentru ca datele să nu fie alterate între momentul citirii și cel al folosirii. Varianta optimistă a acestei blocări se bazează pe presupunerea conform căreia mai multe

tranzacții pot fi executate simultan fără ca una dintre ele să altereze datele alteia, în timp ce varianta pesimistă presupune blocarea unor resurse pe termen mai lung. Fig. 2 descrie blocarea optimistă în varianta ei fundamentală.

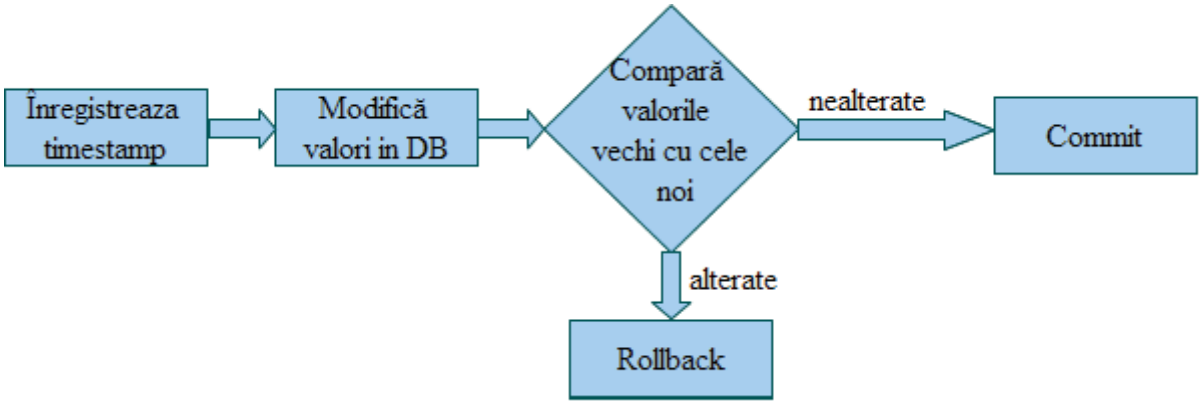

Fig. 2. Blocarea optimistă a resurselor

Într-o aplicație pentru care scalabilitatea și concurența reprezintă cerințe non-funcționale importante se pune problema implementării concurenței cu "optimistic control", deoarece aceasta funcționează bine pentru scenarii cu citiri multiple și modificări rare. Din acest punct de vedere, într-o implementare care folosește JDBC API pur, fără ORM, responsabilitatea verificării integrității obiectelor încărcate și care urmează a fi manipulate revine programatorului. În acest sens trebuie o verificare a versiunii obiectelor care poate fi făcută manual, însă doar în cazul scenariilor simpliste. Pentru scenariile complexe această sarcină devine dificilă întrucât se ajunge la grafuri de obiecte greu de urmărit și controlat.

În aceasta situație sunt dificil de implementat manual șabloane de tranzacții precum:

- sesiune per conversație(„session-per-conversation”);
- sesiune per cerere („session-per-request”);
- sesiune per cerere cu obiecte detașate („session-per-request-with-detached-objects”).

Pe de altă parte, folosirea unui ORM simplifică această sarcină. Spre exemplu, implementarea de Hibernate oferă implicit verificarea automată a versionării atât pentru sesiuni extinse cât și pentru instanțe detașate. Hibernate abordează problema concurenței cu optimistic control folosind versionare automată, obiecte detașate și sesiuni extinse. Vom detalia în cele ce urmează fiecare dintre mecanisme.

Versionare automată. Versionarea implementată de Hibernate folosește un număr de versiune sub forma unui întreg sau al unui timestamp pentru a detecta conflictele de actualizare ale obiectului și pentru a evita pierderea de date. O proprietate a obiectului este marcată ca și proprietate de versionare, iar valoarea ei este incrementată de câte ori se observă că obiectul are o valoare modificată („dirty”).

Obiecte detașate. În cazul obiectelor detașate (obiecte neatașate unei sesiuni) acestea devin atașate diferitelor sesiuni, și la fel ca și în cazul sesiunilor extinse, versiunea lor este verificată la momentul sincronizării stării obiectelor persistente din memorie cu starea din bază de date („flush”), urmând a fi aruncată o excepție în cazul în care este detectată o modificare concurentă.

Sesiuni extinse. În cazul sesiunilor extinse, Hibernate verifică versiunea obiectelor la momentul sincronizării stărilor obiectelor, lansând o excepție dacă este detectată o modificare concurentă.

Performanța

Unul dintre cele mai puternice argumente în defavoarea ORM-urilor este cel conform căruia interogările SQL generate de instrumentele de mapare au o eficiență redusă în multe dintre cazuri în comparație cu interogările SQL scrise manual.

Din punct de vedere al performanței interogărilor, la o analiză mai generală care nu se concentrează strict pe interogări, se poate observa că numărul punctelor critice nu sunt atât de multe încât să influențeze performanța generală a aplicației. În punctele cheie se poate interveni ulterior cu un cod mai eficient (eventual manual), dar cu atenție la partea de mentenanță. Separarea logicii de aducere a datelor aduce un avantaj în cazul în care trebuie efectuate schimbări, numărul acestora devenind mai redus datorită duplicării reduse a codului.

Pe parcursul dezvoltării aplicației se poate evita optimizarea prematură a interogărilor. Utilizarea unui framework ORM duce la scrierea mai rapida a codului, o lizibilitate mai bună, iar duplicările de cod sunt mai ușor de evitat. Odată cu creșterea numărului de interogări se poate folosi un instrument de verificare a vitezei de execuție pentru a identifica punctele sensibile ale performanței și la diferite intervale de timp se poate interveni manual acolo unde e nevoie. De asemenea, printre cele mai la îndemână acțiuni care pot fi luate pentru optimizarea performanței e folosirea stocării în memorie a datelor și indexarea bazei de date. Este de așteptat ca performanța să fie acceptabilă, iar beneficiile folosirii unui mecanism ORM să depășească problemele de performanță.

Maparea entităților

Performanța unui mecanism ORM și productivitatea programatorului depind mult de o bună mapare a entităților la tabelele bazei de date. Printre riscurile unei mapări greșite se numără:

- generarea interogărilor neperformante;
- supraîncărcarea memoriei cu obiecte care nu sunt necesare în urma execuţiei interogărilor;
- productivitate sub așteptări a programatorului din pricina scenariilor complexe determinate de mapări.

Folosirea eficientă a unei implementări ORM presupune o cunoaștere minimă a modului în care implementarea respectivă funcționează „în interior” pentru a evita riscurile descrise mai sus. Un scenariu obișnuit cu risc ridicat constă din însumarea următoarelor condiții:

- cuplare strânsă între entități și existența atributelor de tip colecție de entități;

- relații de 1-n, m-n sau n-1 între entități;
- cascadare pe mai mult de un nivel;
- lipsa unei optimizări din punct de vedere a obținerii datelor („fetching”).

Odată ajuns într-un astfel de scenariu și în situația de a repara o problemă, șansele de apariţie a unor efecte secundare în alte părţi ale proiectului cresc. Testele ajută programatorul să observe dacă soluția găsită creează o problemă într-o alta zona a aplicației, însă provocarea consta în a găsi soluția care face faţă tuturor îngrădirilor datorate mapărilor și nu strică o altă parte din proiect, ceea ce este dificil. De aceea, în procesul de stabilire a entităților care urmează să fie mapate programatorul trebuie sa fie concentrat pe principiul de a menține maparea simplă. Mapările simple și relațiile cat mai simple între entități duc la o întreţinere mai ușoară pe parcurs datorită constrângerilor mai relaxate.

Stocarea datelor în memorie

Stocarea datelor în memorie („caching”) reprezintă o modalitate de creștere a performanței aplicației, principalul său obiectiv fiind de a stoca anumite date în memorie, evitând astfel interogările pe baza de date pentru datele respective.

Folosirea extensivă a reflexiei

Un alt argument care este vehiculat în defavoarea framework-urilor ORM este cel conform căruia folosirea reflexiei („reflection”) în mod intensiv cauzează probleme de performanță. Într-adevăr, conform documentației Oracle anumite operațiuni de optimizare făcute de mașina virtuală nu pot fii efectuate deoarece reflecția implică tipuri rezolvate dinamic, astfel viteza operațiilor efectuate în acest mod este mai redusă, iar recomandarea este de a le evita dacă sunt repetitive în cadrul aplicațiilor sensibile din punct de vedere al performanței. Problema care se pune este în ce măsură aceasta încetinește aplicația și cât de avantajos este compromisul între viteza și flexibilitatea oferită de accesul obiectelor în acest mod. Având în vedere pe de-o parte timpii de execuție ai operațiilor prin reflexie, iar pe de altă parte faptul că majoritatea problemelor legate de performanța în ceea ce privește comunicarea cu baza de date se datorează în mare parte unor deficiențe de arhitectură a acesteia sau a modului greșit de a o interoga, operațiile prin reflexie nu contribuie în mod semnificativ la reducerea performanței. Se poate lua însă în calcul o decizie de optimizare pentru aplicațiile cu o performanță bună, unde în urma testelor se observă că timpii pentru aceste apeluri reprezintă un procent important din timpul necesar comunicării cu baza de date. Operațiile prin reflexie tind să devină tot mai eficiente fiind susținute și de dezvoltarea instrumentelor de manipulare de bytecode precum Java Assist sau ASM. Un semnal important în acesta direcție îl constituie adoptarea bibliotecii Java Assist de către Hibernate pentru eficientizarea acestui tip de operații și folosirea bibliotecii ASM de instrumente precum Oracle TopLink, Apache OpenEJB, AspectJ și multe altele, atât pentru eficientizarea operațiilor de reflexie cat și a programării orientată pe aspecte.

Mentenanța

Întreținerea softului reprezintă un capitol extrem de important indiferent care dintre soluții ar fi aleasă și constituie unul dintre subiectele principale discutate atunci când trebuie făcută o alegere între a folosi un ORM sau nu. O interogare SQL eficientă aduce performanță, însă trebuie observat compromisul făcut din punct de vedere al mentenanței. În general, programatorii sunt relativ familiari cu limbajul SQL și cu bazele de date relaționale, astfel încât primul gând e să meargă direct pe această direcție.

Utilizarea interogărilor SQL pentru comunicarea cu baza de date rezultă într-o cantitate foarte mare de cod folosit doar pentru operații de tip CRUD, necesitând mult timp pentru a fi scris, iar modificările claselor de model îl afectează direct, ducând la o mentenanță dificilă.

JDBC API și SQL oferă o abordare de tip comandă pentru a manipula datele în baza de date. Astfel, tablele și coloanele implicate într-o manipulare de date trebuie specificate de mai multe ori (insert, select, update), ducând la o scădere a flexibilității, la o cantitate mai mare de cod și la o creștere a timpului de implementare. De asemenea, scrierea manuală a codului SQL induce o dependență serioasă între structura tabelelor și codul respectiv. Orice schimbare într-una dintre parți are consecințe imediate în cealaltă parte, ducând în final la o mentenanță dificilă a aplicației și compromisuri lipsite de eleganța în ceea ce privește codul. Integrarea instrumentelor ORM cu modul de programare orientat pe obiecte duce la dezvoltarea unui design general în cadrul căruia codul de acces al datelor are locul lui, alcătuind adesea o componentă separată. Aceasta duce la evitarea codului duplicat, crește potențialul de reutilizare, iar programatorul are posibilitatea de a interacționa cu un graf de obiecte, de a folosi moștenirea, polimorfismul, șabloane de design și alte practici eficiente corespuzătoare programării orientate pe obiecte. Pentru aplicații mici, în care domeniul de business este unul simplu, s-ar putea ca modul de interogare mai apropiat de baza de date să fie mai eficient, în sensul de a folosi JDBC API și de a renunța la efortul de a folosi un ORM. Pentru o aplicație mică mentenanța e mai simplă iar cantitatea de cod e de așteptat să fie redusă.

Metode şi materiale aplicate

API-ul de JDBC („Java Database Connectivity”) oferă acces universal la date din limbajul Java și este compus din două pachete: java.sql și javax.sql. Soluțiile existente pentru stocarea datelor folosind JDBC API sunt restrânse, se poate aminti interfața javax.sql.rowset.CachedRowSet reprezentând un container de înregistrări pe care și le stochează în memorie. Se constituie totodată o componentă JavaBeans ce dispune de scrolling și posibilități de actualizare și serializare. Implementarea de bază acceptă preluarea de date dintr-un set de rezultate, dar poate fi extinsă pentru a obține date și din alte surse tabelare. Cu un astfel de obiect se poate lucra în modul deconectat de baza de date, fiind nevoie de o conexiune doar atunci când datele trebuie sincronizate.

- Pe baza JDBC API s-au dezvoltat și alte biblioteci care oferă mai multe soluții de stocare a datelor în memorie, astfel de exemple fiind:
- Extensia Oracle pentru JDBC - oferă o interfață și implementare pentru statement cache (stocarea statement-urilor care sunt folosite în mod repetat)
- Implementarea PostgreSQL pentru JDBC - oferă un wrapper de stocare a statement-urilor peste implementarea de bază a JDBC-ului.

Există și alte soluții de stocare în memorie oferite de părți terțe care pot fi integrate într-o aplicație care folosește direct JDBC API. Astfel de exemple sunt:

- Commons JCS
- Ehcache

Din punct de vedere al framework-urilor ORM, acestea completează discuția despre „a nu discuta cu baza de date" cu subiectul „a nu discuta cu API-ul JDBC" în cazul Java, în sensul de a face transparentă programatorului relația cu API-ul JDBC, și de a manipula obiectele stocate în memorie doar la nivelul ORM-ului și al implementărilor sale pentru astfel de stocări.

Hibernate oferă primul nivel de stocare („first-level cache„) care e asociat cu sesiunea și e folosit implicit; mai oferă un al doilea nivel („second-level cache") care e asociat cu fabrica de sesiuni și e opțional (Fig. 3).

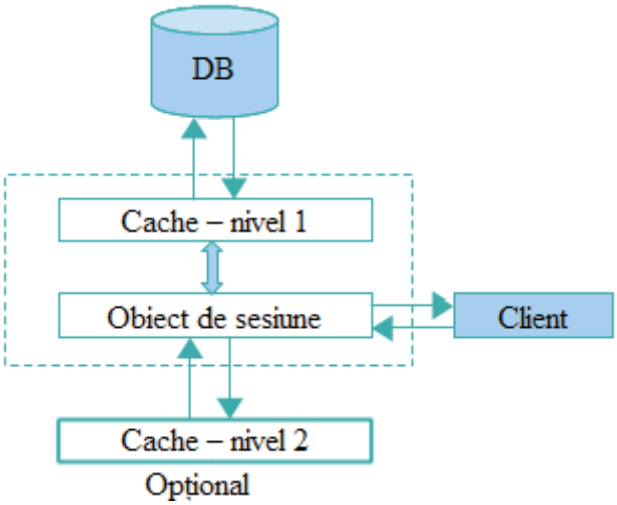

Fig. 3. Nivelele unu și doi de stocare a datelor oferit de Hibernate

Primul nivel de stocare e folosit pentru a reduce numărul interogărilor pe baza de date la nivelul unei sesiuni [1]. Datele nu pot fi stocate în memorie și folosite de alte sesiuni în afara celeia care a adus datele inițial. Cel de-al doilea nivel de stocare este localizat la nivelul fabricii de sesiuni și este capabil de a depozita date din diferite sesiuni. Acest lucru înseamnă că toate obiectele de sesiune pot accesa aceleași date stocate. Hibernate suportă trei implementări „open source" pentru folosirea nivelului 2 de stocare. Acestea sunt:

- Ehcache

- OSCache
- JBoss TreeCache

Prin faptul că primul nivel de cache e folosit în mod implicit, Hibernate aduce deja un plus important în ceea ce privește performanța din punct de vedere al comunicării cu baza de date față de folosirea API-ul de JDBC. De asemenea, cel de-al doilea nivel de stocare, folosit în general pentru optimizarea aplicațiilor relativ bune din punct de vedere al performanței, duce la o creștere importantă a acesteia, iar programatorul are flexibilitatea de a alege implementarea care se potrivește cel mai bine contextului dat.

Rezultate obținute

În figurile de mai jos putem observa aplicarea ORM-ului Hibernate prin intermediul limbajului Java. Prin intermediul paradigmei POO a fost declarată o entitate „Files” cu atributele necesare (Figura 4).

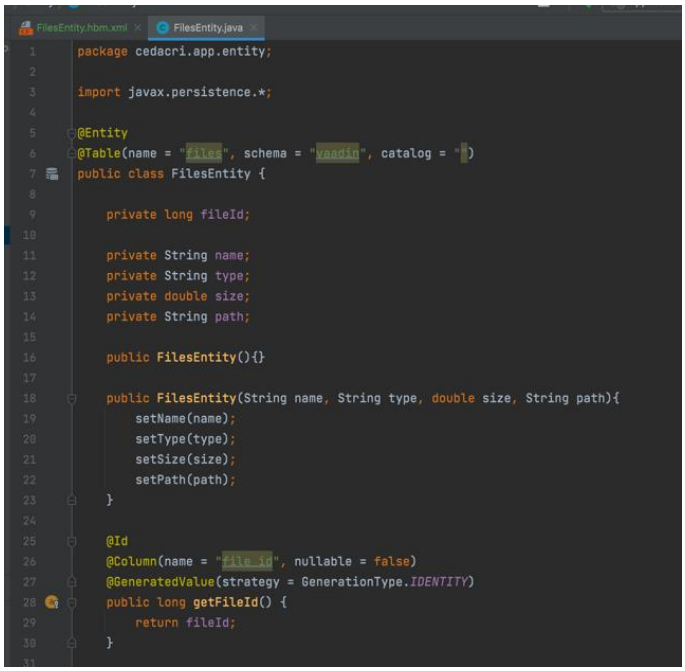

Fig. 4. Entitatea „Files” reprezentată ca clasă

În fișierul XML (Fig. 5) sunt mapate fiecare atribut a entității Files cu parametrii necesari pentru identificarea tipurilor de date care vor fi depistate de SQL.

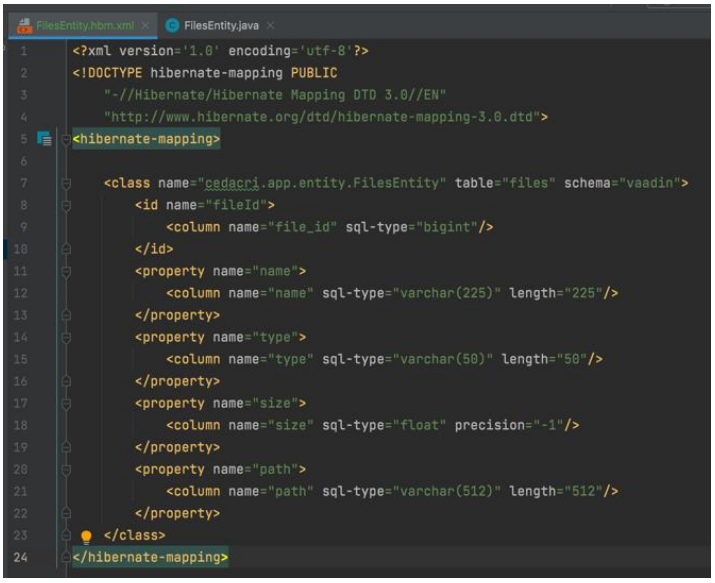

Fig. 5. Entitatea „Files” mapată în XML

La executarea programului Java, Hibernate creează o conexiune la baza de date SQL și generează entitățile conform fișierului XML cu tipurile de date corespunzătoare (Fig. 6).

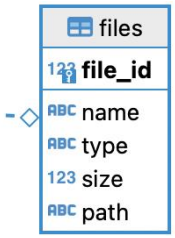

Fig. 6. Entitatea „Files” reprezentată în SQL

Avantaje	Dezavantaje
Modelarea domeniului real	Timp investit în învățarea unui framework ORM
Reducerea cantității de cod	Dificultăți în definirea mapărilor
Mentenanță simplificată	Performanța interogărilor generate
Noi limbaje pentru interogări	Lipsa de control asupra interogărilor generate
Abordări eficiente a problemelor de concurență	Pe termen scurt nu aduce un salt important din punct de vedere al productivității
Abordări eficiente pentru stocarea datelor în memorie	
Productivitate pe termen mediu și lung	

Fig. 7. Avantajele și dezavantajele de utilizare a tehnicii ORM

Concluzii

Folosirea unui framework ORM și alegerea lui sunt decizii care trebuie luate în cunoștință de cauză și care depind de specificul proiectului. Dificultatea inițială în dezvoltarea unui cod bazat pe un framework ORM constă în curba de învățare a framework-ului respectiv, urmată fiind de dificultatea mapării datelor la coloanele tabelelor, o operație care trebuie făcută manual ținând cont

și de modul în care aceste relații vor defini în final structura tabelelor și a coloanelor. Curba de învățare ar trebui (cel puțin teoretic) să fie răsplătită pe termen lung, mai ales în cazul proiectelor medii și mari. Odată înțeles mecanismul ORM, timpul de dezvoltare ar urma să scadă ducând la o productivitate îmbunătățită a programatorului (Fig.7).

Pentru a folosi eficient un framework ORM nu ajunge experiența dobândită într-un limbaj de programare orientat pe obiecte, e necesară cunoașterea modelului relațional și a limbajului SQL. Framework-urile ORM permit evitarea scrierii de cod repetitiv și creșterea productivității, însă doar prin cunoașterea detaliată a framework-ului folosit el poate fi întrebuințat în mod optim, iar cunoașterea limbajului SQL ajută programatorul în cazul problemelor importante de performanță. Scopul final este acela de a avea productivitate și performanță în managementul datelor persistente.

Bibliografie

1. https://hibernate.org/orm/documentation/5.5/