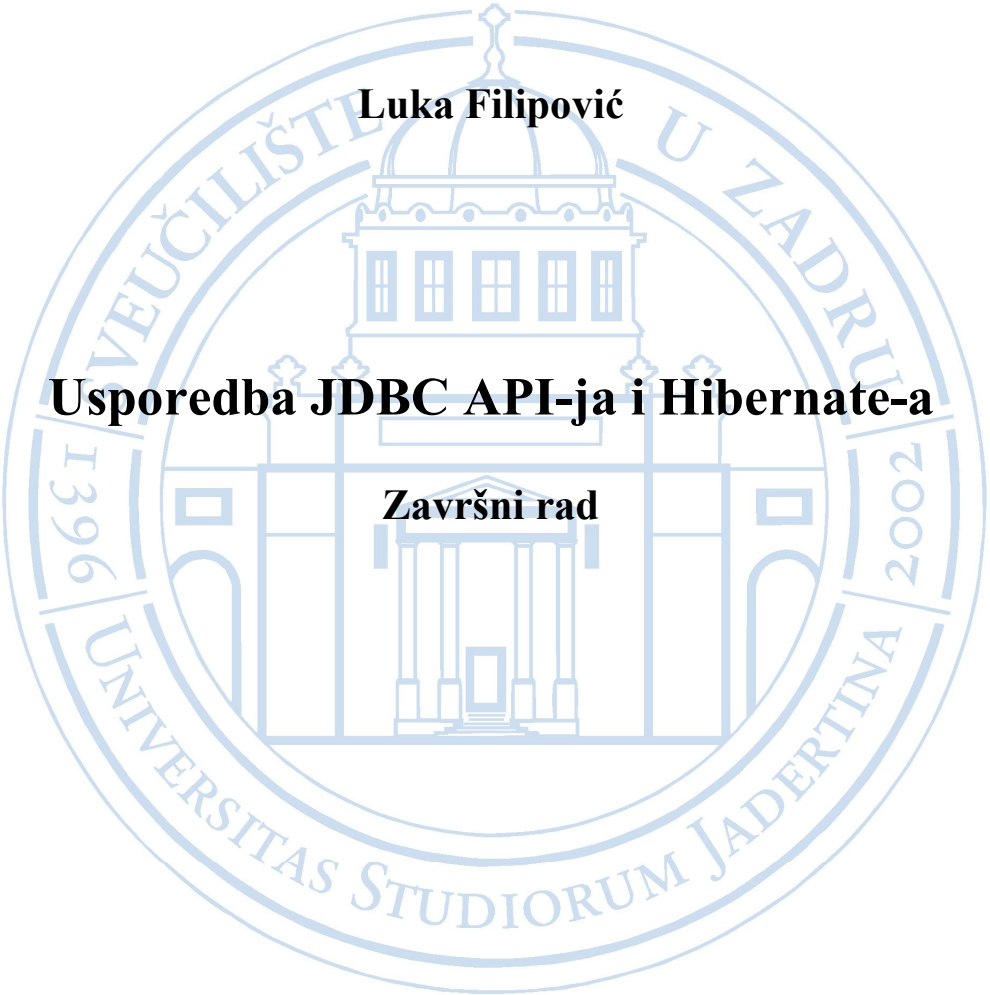
Usporedba JDBC API-ja i Hibernate-a

Filipović, Luka

Sveučilište u Zadru

Stručni prijediplomski studij Informacijske tehnologije

Luka Filipović



Usporedba JDBC API-ja i Hibernate-a

Završni rad

Zadar, 2023.

Sveučilište u Zadru

Stručni prijediplomski studij Informacijske tehnologije

Usporedba JDBC API-ja i Hibernate-a

Završni rad

Student:
Luka Filipović

Mentor:
doc. dr. sc. Ante Panjkota

Zadar, 2023.

Izjava o akademskoj čestitosti

Ja, **Luka Filipović,** ovime izjavljujem da je moj **završni** rad pod naslovom **Usporedba JDBC API-ja i Hibernate-a** rezultat mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na izvore i radove navedene u bilješkama i popisu literature. Ni jedan dio mojega rada nije napisan na nedopušten način, odnosno nije prepisan iz necitiranih radova i ne krši bilo čija autorska prava.

Izjavljujem da ni jedan dio ovoga rada nije iskorišten u kojem drugom radu pri bilo kojoj drugoj visokoškolskoj, znanstvenoj, obrazovnoj ili inoj ustanovi.

Sadržaj mojega rada u potpunosti odgovara sadržaju obranjenoga i nakon obrane uređenoga rada.

Zadar, 8. lipnja 2023.

Sažetak

U radu je prikazana usporedba JDBC API-ja kao osnovnog API-ja za komunikaciju Java aplikacije s bazom podataka i Hibernate-a kao najpoznatijeg predstavnika ORM frameworka. ORM frameworci razlikuju se od JDBC API-ja ponajviše po tome što pružaju mogućnost automatskog mapiranja objekata u tablice u bazi i obrnuto. To je ujedno i glavna prednost Hibernate-a u odnosu na JDBC API. Rad je podijeljen u dva dijela. U prvom dijelu teoretski su opisani JDBC API i Hibernate te njihov razvoj, arhitektura i komponente kako bi drugi dio rada gdje je prikazana njihova usporedba bio razumljiviji. Prilikom usporedbe korišteni su primjeri koji su djelo autora ovoga rada. Usporedba je rađena kako bi se vidjele razlike u njihovoj konfiguraciji, spajanju na bazu, implementaciji entiteta, kreiranju veza između entiteta, upravljanju iznimkama te načinu izvođenja osnovnih CRUD operacija nad bazom podataka. Na kraju je prikazana i usporedba njihovih performansi te opisan caching mehanizam koji Hibernate koristi s ciljem povećanja performansi.

Ključne riječi: JDBC API, Hibernate, ORM, baza podataka, aplikacija, objekt, tablica, entitet

Popis korištenih kratica

JDBC	Java DataBase Connectivity = Java API za povezivanje s bazom podataka, izdavanje upita, izvođenje naredbi nad bazom i upravljanje skupovima rezultata dobivenih iz baze.
API	Application Programming Interface = aplikacijsko programsko sučelje
ORM	Object-Relational Mapping = tehnika u programiranju za pretvaranje podataka iz baze u objekte u objektno-orijentiranim programskim jezicima i obratno
SQL	Structured Query Language = skriptni jezik za upravljanje relacijskim bazama podataka
ODBC	Open DataBase Connectivity = standardni API za pristup sustavima za upravljanje bazama podataka
JDK	Java development kit = distribucija Java tehnologije
SE	Standard Edition = SE u Java SE označava osnovno izdanje
CRUD	Create, read, update, delete = kreiraj, čitaj, izmjeni, izbriši - operacije nad bazom podataka
HTML	HyperText Markup Language = opisni jezik za izradu web stranica
RDBMS	Relational Database Management System = sustav za upravljanje relacijskim bazama podataka
HQL	Hibernate Query Language = objektno orijentirani jezik za slanje upita prema bazi podataka
EJB	Enterprise Java Beans = Java API za modelarnu konstrukciju poslovnog softvera
JTA	Java Transaction API = Java API koji omogućava obavljanje distribuiranih transakcija
JNDI	Java Naming and Directory Interface = sučelje za imenovanje u programskom jeziku Java

J2EE	Java 2 platform Enterprise Edition = standardna Java platforma za razvoj poslovnih aplikacija
POJO	Plain Old Java Object = običan Java objekt koji nema nikakve restrikcije

Sadržaj

Uvod

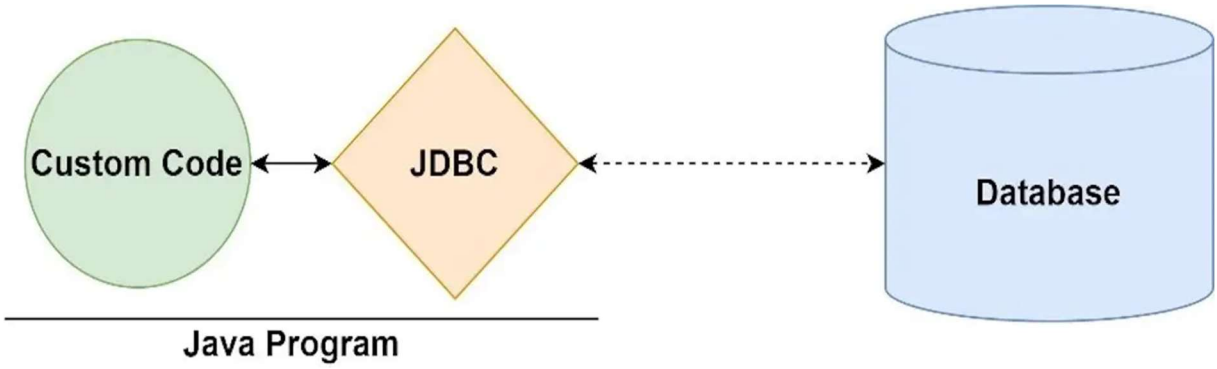
Velik broj današnjih aplikacija rade na principu pohranjivanja podataka u neku bazu podataka. Kako bi se razvojnim programerima olakšao razvoj takvih aplikacija, u programskim jezicima postoje mnogi alati i tehnologije koje služe za integraciju s bazom podataka te olakšavaju razvoj takvih aplikacija.

U programskom jeziku Java, osnovni alat koji se koristi za pristup i rad s bazom podataka je JDBC API (Java Database Connectivity Application Interface). Napretkom same Jave, razvijali su se i drugi alati koji nude više mogućnosti u radu s bazama podataka. Razvoj takvih alata bazirao se na omogućavanju mapiranja entiteta u objekte u Javi i obrnuto. Na taj način smanjuje se količina koda i ubrzava se sam postupak razvoja aplikacije. To je dovelo do razloga mnogih ORM (Object Relational Mapping) alata i radnih okvira (eng. framework, u daljnem tekstu framework). Najpopularniji među njima je Hibernate.

Cilj ovog rada je usporediti mogućnosti JDBC API-ja kao osnovnog i standardnog API-ja za komunikaciju Java aplikacije s bazom podataka i Hibernate-a kao najpopularnijeg predstavnika „modernih“ ORM frameworka. Rad je podijeljen u dva dijela. U prvom dijelu bit će teoretski opisani JDBC API i Hibernate, njihov razvoj te objašnjena njihova arhitektura i njihove komponente. Već će se tu moći uvidjeti neke razlike između ova dva alata, a konkretna njihova usporedba bit će prikazana u drugom dijelu rada. Usporedba je temeljena na primjerima koje je izradio sam autor rada, a osim što će se uspoređivati mogućnosti ova dva alata, pomoću primjera bit će prikazano i kako se pojedine operacije nad bazom izvode pomoću jednog, a kako pomoću drugog alata.

1. JDBC API

JDBC skraćenica je od Java Database Connectivity, a predstavlja API u programskom jeziku Java koji služi za povezivanje s bazom podataka, izdavanje upita (eng. query) i izvođenje naredbi nad bazom kao i upravljanje skupovima rezultata dobivenih iz baze [1]. Jednostavnije rečeno, JDBC API koristimo za interakciju Java programa s bazom podataka te ga možemo zamisliti kao most između našeg koda i baze.



Slika 1. JDBC kao most između Java programa i baze podataka

(Izvor: https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html)

Osnovne funkcionalnosti koje JDBC API pruža su: [2]

- Stvaranje konekcije s bazom podataka
- Slanje upita i izdavanje naredbi prema bazi
- Pregled i obrada rezultata dobivenih iz baze.

JDBC API upite i naredbe prema bazi šalje koristeći SQL te podržava sva SQL narječja (eng. SQL dialects). Također, kako Fisher, Ellis i Bruce [2] navode JDBC API nadilazi sam SQL omogućavajući interakciju i s drugim izvorima podataka kao što su datoteke s tabličnim podacima. Kombinacija Jave i JDBC API-ja programerima omogućava da napišu jedan Java program i putem JDBC API-ja šalju upite prema bilo kojoj bazi podataka putem bilo koje platforme (koristeći Java Virtual Machine).

Na razini Jave kao programskog jezika, JDBC nije ništa drugo nego biblioteka (eng. library) koja se sastoji od skupa klasa (eng. class) i sučelja (eng. interface). Kako Bales [3] navodi JDBC

API baziran je primarno na sučeljima, a jedina konkretna klasa je *DriverManager*. Sučelja i klase JDBC-a bit će detaljnije opisana u nastavku rada.

1.1 Razvoj JDBC API-ja

Prije nastanka JDBC-ja, postojao je ODBC odnosno Open Database Connectivity. Razvio ga je Microsoft davne 1992. godine kao API za pristup SQL bazama podataka. Industrija ga je prihvatila kao standard za pristup bazama u Windows okruženju. Međutim, u „Java svijetu" nije bio dobro prihvaćen prvenstveno zbog toga što je napisan u C programskom jeziku pa je korištenje u Javi zahtijevalo dodatan API [4]. Osim toga, ODBC ovisan je o platformi, odnosno o Windowsu, a znamo kako je Java neovisna o platformi. Zbog toga, došlo je do potrebe za razvojem JDBC-ja kao API-ja primarno namijenjenog Java programskom jeziku. Osim same ideje, JDBC iz ODBC-a vuće inspiraciju i u arhitekturi.

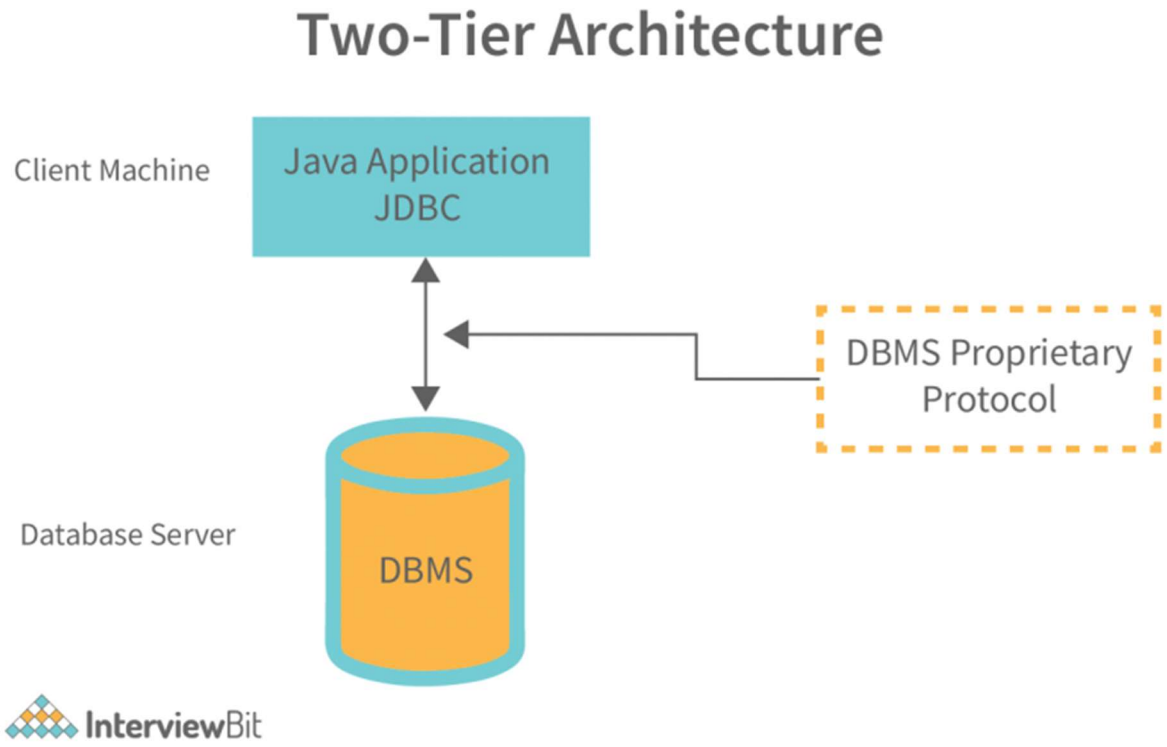
JDBC je jedna od prvih biblioteka (eng. library) razvijenih za Javu, a izdao ju je Sun Microsystems 1997. godine kao dio Java JDK-ja 1.1. JDBC je prvenstveno zamišljen kao API na strani klijenta (eng. client-side). To se promijenilo dolaskom verzije 2.0 kada je u biblioteku dodan novi paket (*javax.sql*, uz dotadašnji *java.sql*), a koji je donio podršku za konekcije s bazom na strani poslužitelja (eng. server-side). Iduće verzije nosile su ažuriranja za oba paketa, a najnovija verzija u trenutku pisanja ovoga rada odnosno verzija JDBC 4.3, izdana je 2017. kao dio Java SE 9 [1].

1.2 Arhitektura

Arhitektura JDBC API-ja podržava dvoslojni i troslojni model za komunikaciju s bazom podataka [2].

1.2.1 Dvoslojna arhitektura

U dvoslojnoj arhitekturi Java program komunicira izravno s bazom podataka za što koristi JDBC pogonitelj (eng. driver, u daljnem tekstu driver). Korisnikove naredbe izvode se izravno nad bazom podataka, a rezultati tih naredbi vraćaju se korisniku. Ova arhitektura podržava klijent/poslužitelj (eng. client/server) arhitekturu na način da je klijent korisnik, odnosno njegov uređaj, a poslužitelj je uređaj na kojem se nalazi baza podataka [2].

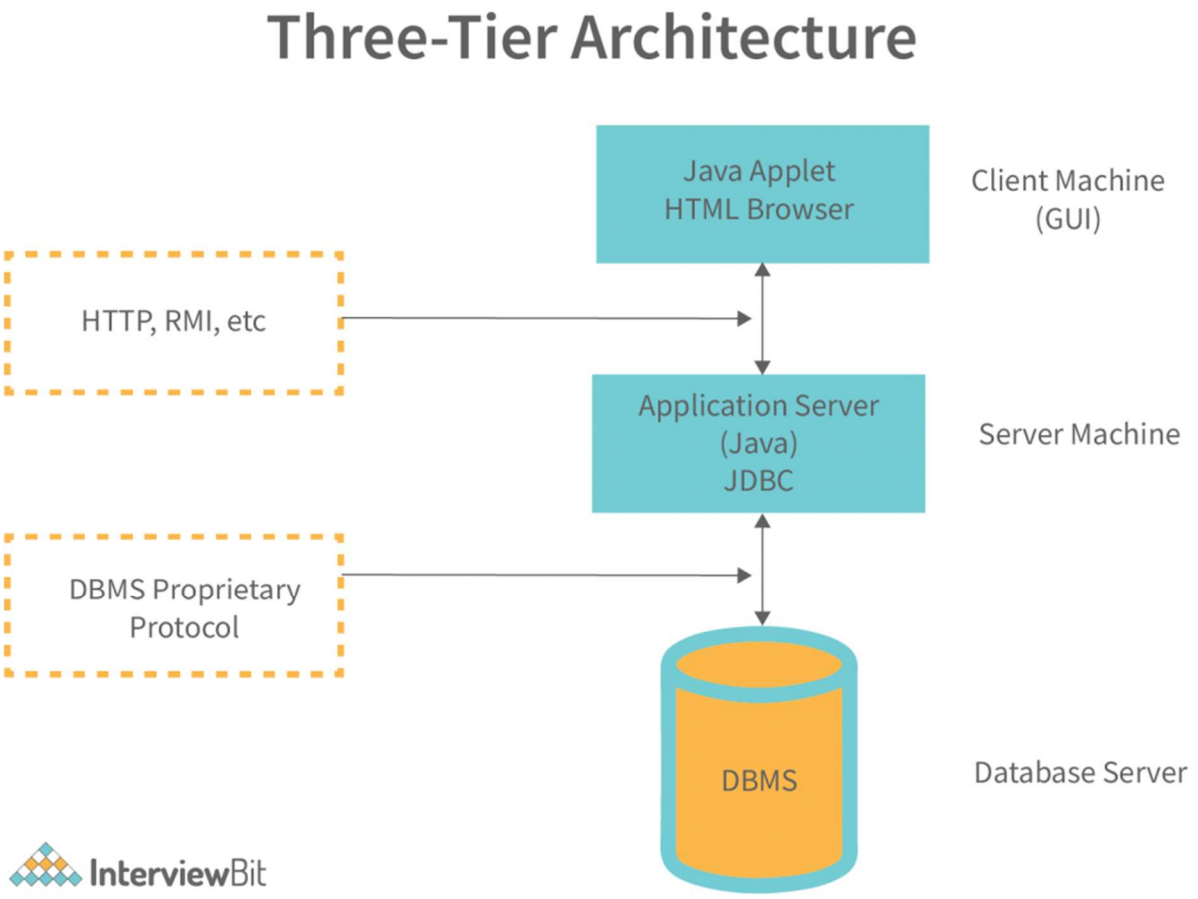


Slika 2. Dvoslojna arhitektura

(Izvor: *https://www.interviewbit.com/blog/jdbc-architecture/*)

1.2.2 Troslojna arhitektura

U troslojnoj arhitekturi korisnik, putem npr. HTML preglednika, šalje zahtjev Java programu koji u ovoj arhitekturi predstavlja srednji sloj, a taj srednji sloj zatim šalje naredbe prema bazi podataka. Rezultati tih naredbi opet se šalju srednjem sloju, a zatim ih srednji sloj šalje korisniku. Dakle, nema izravne komunikacije. Prednosti ove arhitekture su jednostavniji razvoj i bolje performanse [2].

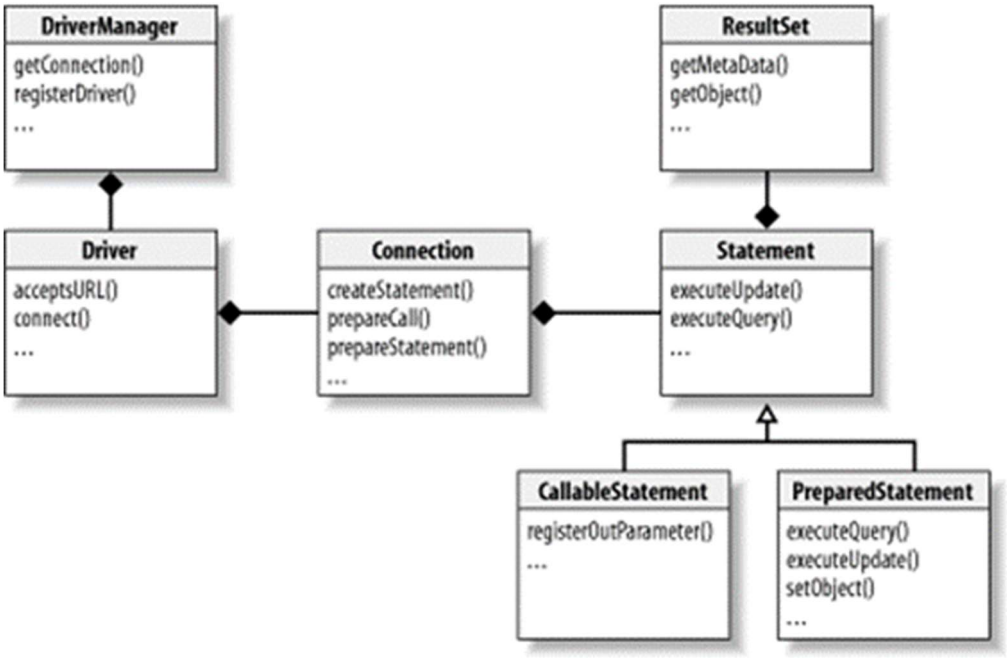


Slika 3. Troslojna arhitektura

(Izvor: *https://www.interviewbit.com/blog/jdbc-architecture/*)

1.3 Komponente JDBC-ja

Na početku rada spomenuto je kako je JDBC u Javi zapravo biblioteka sa svojim sučeljima i klasama. Ta sučelja i klase čine osnovne komponente JDBC-ja. Na slici 4. možemo vidjeti dijagram klasa koji prikazuje te komponente, a u ovom poglavlju bit će teoretski opisano čemu pojedina komponenta služi, dok će njihovo konkretno korištenje biti vidljivo u primjerima koji će se nalaziti u drugom dijelu rada.



Slika 4. Dijagram klasa JDBC biblioteke

(Izvor: *https://www.oreilly.com/library/view/javaserver-pages-3rd/0596005636/ch24s01.html*)

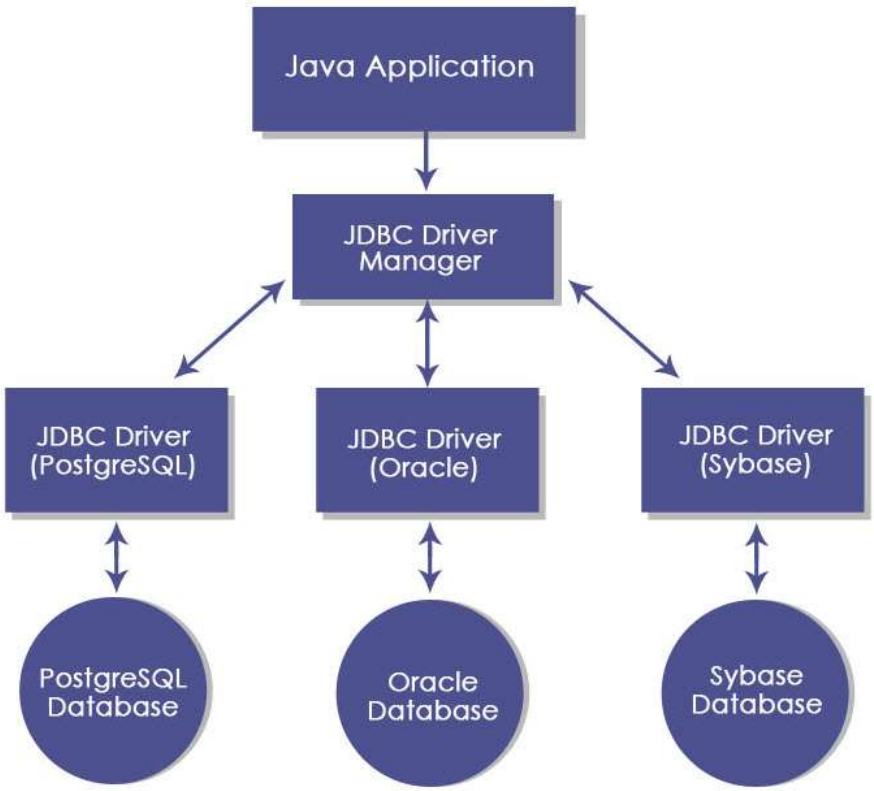
1.3.1 DriverManager

DriverManager je klasa koja upravlja driverima baza podataka odnosno osigurava da se koristi ispravni driver za svaku bazu podataka. *DriverManager* zapravo uparuje zahtjev za povezivanjem iz Java programa sa odgovarajućim driverom baze podataka [5].

1.3.2 Driver

Driveri su sučelja koja upravljaju komunikacijom sa poslužiteljem baze podataka. U praksi se sa driverima upravlja veoma rijetko, već se sve obavlja putem *DriverManager*-a kako je opisano u prethodnom poglavlju [5].

Sada kad znamo čemu služe *DriverManager* i *Driver* možemo se vratiti na poglavlje o arhitekturi i pogledati opisana dva modela arhitekture JDBC-ja. Na modelima, između Java programa i baze podataka možemo dodati *DriverManager* i odgovarajuće drivere baza podataka pa arhitekturu detaljnije možemo prikazati na način kako je prikazano na sljedećoj slici.



Slika 5. Arhitektura JDBC-ja sa DriverManager-om i driverima

(Izvor: *https://www.educba.com/jdbc-architecture/)*

1.3.3 Connection

Connection je sučelje sa svim metodama za „kontaktiranje“ baze podataka. Sva komunikacija s bazom podataka odvija se preko *Connection* objekta [5].

1.3.4 Statement

Statement je sučelje čiji se objekti koriste za izvođenje SQL upita nad bazom. Također, razlikujemo *PreparedStatement* i *CallableStatement* sučelja koja nasljeđuju *Statement* sučelje.

PreparedStatement sučelje koristimo za izvođenje unaprijed pripremljenih SQL upita što omogućuje brzo i učinkovito izvršavanje upita nad bazom. Još važnije, to nam omogućava i izvođenje dinamičkih upita sa parametrima [6].

CallableStatement koristi se za izvođenje spremljenih SQL procedura. Te procedure su zapravo skupine naredbi koje sastavljamo u bazi za izvršavanje nekog zadatka. To je korisno kada imamo više tablica s određenim kompleksnim scenarijima, pa nam je, umjesto slanja velikog broja upita prema bazi, lakše poslati samo potrebne podatke i pustiti da spremljene procedure izvrše potrebnu logiku [6] [7].

1.3.5 ResultSet

ResultSet objekti čuvaju podatke dobivene iz baze podataka nakon izvođenja SQL upita u obliku stupaca i redaka. Ti se objekti ponašaju kao iteratori pomoću kojih se možemo kretati po dohvaćenim podacima. Osim toga, *ResultSet* sučelje sadrži metode za dohvaćanje tih podataka kao i za pretvorbu iz SQL tipova podataka u Java tipove podatka [5], [6].

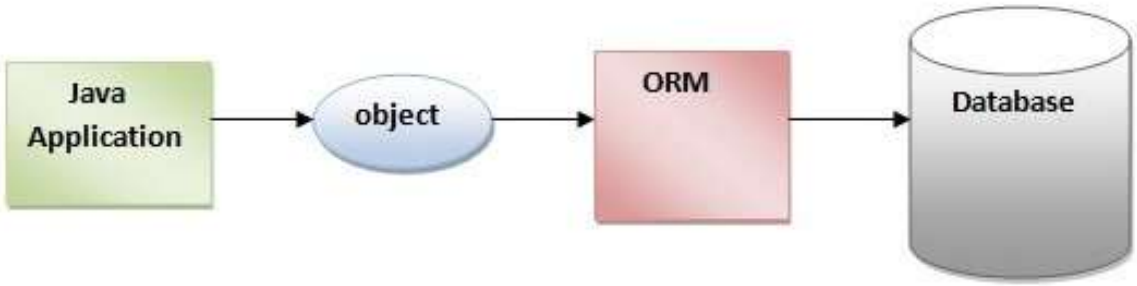
2. ORM frameworci

Kao što je rečeno u uvodu, Hibernate je predstavnik ORM frameworka, pa prije nego što kažemo nešto o Hibernate-u, potrebno je objasniti što zapravo ORM predstavlja i čemu potreba za ORM frameworcima.

ORM je skraćenica od Object-Relational Mapping, a predstavlja tehniku u programiranju za pretvaranje podataka iz baze u objekte u objektno-orijentiranim programskim jezicima i obrnuto [8]. Može se reći da je ORM način usklađivanja programskog koda sa strukturama baza podataka na način da stvara određeni sloj između programskog jezika i baze što uvelike pojednostavljuje interakciju između baza i objektno-orijentiranih programskih jezika [9].

Koristeći ORM programeri mogu izvršavati CRUD operacije (Create, Read, Update and Delete) nad bazom bez da uopće koriste SQL. To smanjuje količinu koda jer programer ne mora pisati SQL upite. Štoviše, programer ne mora ni znati SQL, a može koristeći ORM izvoditi operacije nad bazom. Ostale prednosti ORM-a vidjet ćemo u daljnjem dijelu rada kada će biti prikazana usporedba između JDBC API-ja i Hibernate-a, a važno je reći da ORM uvelike ubrzava i olakšava sam proces razvoja aplikacije zbog čega ga programeri često odlučuju koristiti.

Ako govorimo o programskom jeziku Java, treba napomenuti da svi ORM frameworci „ispod haube“ zapravo koriste JDBC.



Slika 6. ORM u Java programskom jeziku

(Izvor: *https://www.javatpoint.com/hibernate-tutorial*)

Osim Hibernate-a, neki od poznatijih ORM frameworka i alata u Javi su [8]: **Enterprise JavaBeans EntityBeans**, **Java Data Objects**, **Castor**, **TopLink**, **SpringDAO** i mnogi drugi.

3. Hibernate

Hibernate je Java ORM framework otvorenog koda (eng. open-source) koji se koristi primarno u razvoju web aplikacija. Hibernate omogućuje mapiranje Java klasa (objekata) u tablice u bazi podataka kao i Java tipove podataka u SQL tipove podataka. Također omogućuje slanje upita prema bazi i dohvaćanje podataka iz baze [10].

3.1 Hibernate i JPA

JPA odnosi se na Jakarta Persistance API (ranije Java Persistance API), a predstavlja specifikaciju koja se brine o ustrajnosti (upornosti - eng. persistance) Java objekata. Ustrajnost je svojstvo objekta koje mu omogućava da nadživi proces koji ga je stvorio. JPA specifikacija omogućava definiranje objekata koji trebaju biti ustrajni (eng. persisted) i na koji način trebaju biti ustrajni u Java aplikaciji. Sam po sebi, JPA nije ni alat ni framework, već samo definira određena pravila koja programer treba slijediti. Hibernate je, kao i ostali ORM alati, implementacija JPA što znači da koristi JPA specifikaciju i standarde. Međutim, čemu potreba za nečim takvim? Odgovor je vrlo jednostavan. Ukoliko ORM alati prate jednake standarde i koriste istu specifikaciju, vrlo lako možemo aplikaciju koju razvijamo prebaciti s korištenja jednog ORM alata na drugi [11].

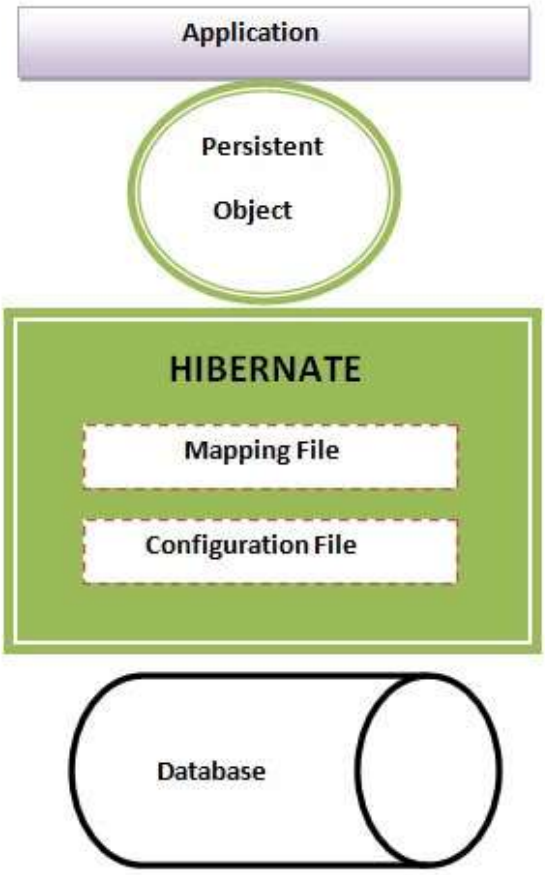
Konkretna implementacija JPA u Hibernate-u odvija se koristeći JPA anotacije prilikom implementacije entiteta u Javi, a Hibernate dalje brine o konfiguraciji prilikom izvođenja operacija nad bazom. Primjer za to bit će prikazan u daljnjem radu kada će biti prikazano kako se izvodi implementacija entiteta koristeći Hibernate.

3.2 Razvoj Hibernate-a

Hibernate je razvio Gavin King 2001. godine, zajedno sa svojim kolegama iz tvrtke CircusTechnologies. Primarna namjena bila mu je pružiti bolje mogućnosti ustrajnosti nego tadašnji EJB2 (Enterprise JavaBeans API) smanjenjem kompleksnosti i dodavanjem značajki koje nedostaju. Gavin King je, osim razvoja Hibernate-a, doprinio i razvoju EJB3 te JPA. 2003. godine izašao je Hibernate 2 koji je sa sobom donio brojna unaprijeđena u odnosu na prvu verziju. Treća verzija, izašla 2005., sa sobom donosi nove značajke poput presretača (eng. interceptors), korisnički definiranih filtera i anotacija za Java JDK 5.0. 2010. godine Hibernate 3, odnosno verzije 3.5 i dalje, implementiraju JPA verziju 2.0. Hibernate 4, izašao 2011, donosi nove značajke poput bolje podrške, boljeg otvaranja Hibernate sesije i mnoge druge. 2012. godine Hibernate se u verziji Hibernate ORM 4.1.9 prvi put pojavljuje sa riječi ORM u nazivu. U prosincu 2012. godine, verzija Hibernate ORM 4.3 dolazi sa podrškom za JPA 2.1. Peta verzija, 2015., sa sobom donosi potpunu podršku za Javu 8 te dodatna poboljšanja. 2019. godine, verzija 5.4 postaje kompatibilna sa JPA 2.2 te pruža kompatibilnost za Javu 11, a u verziji 5.5 Hibernate postaje kompatibilan s novom verzijom JPA. Točnije, verzijom 3.0 kada je JPA postao Jakarta Persistance API umjesto dotadašnjeg Java Persistance API-ja. Hibernate 6 izašao je 2022. godine, a trenutna verzija za vrijeme pisanja ovoga rada, odnosno verzija 6.2 izašla je 2023. godine te pruža kompatibilnost za Javu 11, 17 i 18 te JPA 3.1 [12] [13].

3.3 Arhitektura

A. A. Puntambekar [14] objašnjava arhitekturu Hibernate-a putem troslojnog modela gdje Hibernate ima ulogu srednjeg sloja između Java aplikacije i baze. Na slici 7. možemo vidjeti prikaz takvog modela.

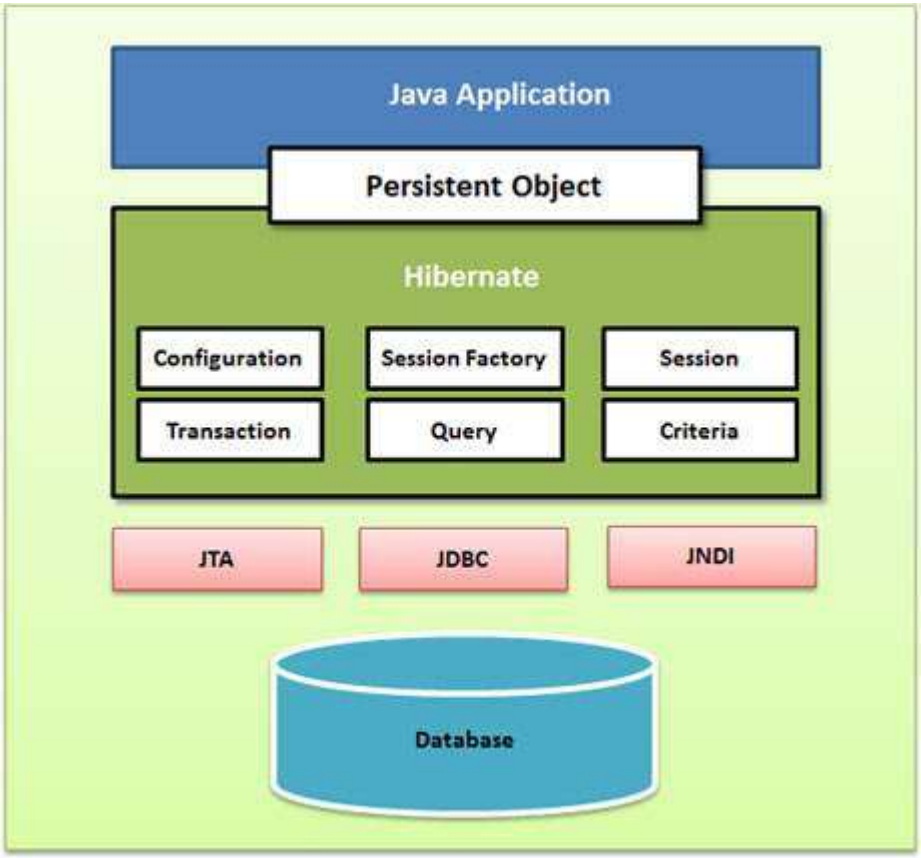


Slika 7. Troslojna arhitektura Hibernate-a

(Izvor: https://www.javatpoint.com/hibernate-architecture)

Kako bi uopće mogli koristiti Hibernate potrebno ga je najprije konfigurirati. To se najčešće vrši pomoću dvije konfiguracijske datoteke: jedna koja služi za konfiguraciju samog Hibernate-a i druga kojom konfiguriramo mapiranje između klasa i tablica u bazi [14].

Neki izvori, poput internetskog izvora JavaTPoint [15], arhitekturu Hibernate-a prikazuju kao četveroslojnu, dodavajući i „Backend API“ sloj između Hibernate sloja i sloja baze podataka. Razlog tomu je što Hibernate koristi neke od postojećih Java API-ja, točnije JDBC API, JTA (Java Tranasction API) i JNDI (Java Naming And Directory Interface). Takav prikaz detaljnije arhitekture vidljiv je na slici 8. Na slici su također vidljive i komponente Hibernate-a, odnosno sučelja i klase koje ga sačinjavaju, a koje će biti detaljnije opisane u idućem poglavlju.



Slika 8. Četveroslojna arhitektura Hibernate-a

(Izvor: https://www.tutorialspoint.com/hibernate/hibernate_architecture.htm)

JDBC u ovom slučaju Hibernate-u pruža mogućnost da podržava svaku bazu podataka putem JDBC drivera. JNDI i JTA omogućavaju Hibernate-u da se integrira sa J2EE poslužiteljima aplikacija [16].

3.4 Komponente Hibernate-a

U ovom poglavlju bit će opisane Hibernate-ove klase i sučelja koja čine komponente Hibernate-a. Preciznije, u ovom dijelu rada teorijski su opisane te komponente, dok će konkretni primjeri korištenja istih biti vidljivi kasnije u radu.

3.4.1 Configuration

Configuration objekt je prvi Hibernate objekt koji treba kreirati u svakoj aplikaciji te se obično kreira samo jednom prilikom inicijalizacije aplikacije. On zapravo predstavlja konfiguraciju Hibernate-a sa već spomenute dvije bitne komponente: konfiguracija konekcije s bazom podataka i konfiguraciju mapiranja između klasa i tablica u bazi. Konekcija s bazom podataka najčešće se konfigurira putem konfiguracijskih datoteka: *hibernate.properties* i *hibernate.cfg.xml* [16].

3.4.2 SessionFactory

Za kreiranje *SessionFactory* objekta potreban je *Configuration* objekt. *SessionFactory* objekt zapravo konfigurira aplikaciju koristeći konfiguracijsku datoteku iz *Configuration* objekta te omogućava da se *Session* objekt može kreirati. Ukoliko koristimo višenitno programiranje, sve niti (eng. threads) koristit će isti *SessionFactory* objekt. Osim toga, *SessionFactory* objekt obično se, kao i *Configuration* objekt, kreira samo jednom prilikom inicijalizacije aplikacije. Jedan *SessionFactory* objekt koristi se za jednu bazu podataka, pa ukoliko u aplikaciji koristimo više baza podataka, trebat ćemo kreirati i više *SessionFactory* objekata, za svaku po jedan [16].

3.4.3 Session

Session objekt koristi se za stvaranje „fizičke“ konekcije s bazom podataka i kreira se svaki put kada se treba izvršiti neka interakcija s bazom. Ustrajni objekti spremaju se i dohvaćaju iz baze putem *Session* objekta [16].

3.4.4 Transaction

Transakcija predstavlja jedinicu rada s bazom podataka i većina RDBMS-a podržavaju rad s transakcijama[1]. U Hibernate-u, transakcijama upravljaju upravitelji transakcijama (eng. transaction manager) iz JDBC-ja ili JTA. *Transaction* objekt je opcionalan u Hibernate-u i umjesto da ga koristimo, transakcijama možemo upravljati koristeći vlastiti kod [16].

[1] Transakcija baze podataka simbolizira jedinicu rada koja se izvodi unutar sustava za upravljanje bazom podataka (RDBMS-a). Transakcija općenito predstavlja bilo koju promjenu u bazi podataka. Može se sastojati od više operacija nad bazom, ali sve operacije trebaju biti izvršene kako bi transakcija bila uspješna.

3.4.5 Query

Query objekti koriste HQL ili čisti SQL za dohvaćanje podataka iz baze i kreiranje objekata. Osim toga *Query* objektom možemo davati i parametre upitu nad bazom te ograničavati broj rezultata upita [16].

3.4.6 Criteria

Criteria objekti koriste se također za dohvaćanje podataka iz baze ali na objektno-orijentirani način. Također, *Criteria* objektima lakše upravljamo vezama između tablica u bazi kao i grupiranjem, paginiranjem te ostalim uvjetima prema kojima želimo dohvatiti podatke iz baze [17].

3.5 HQL

Znamo kako je SQL skriptni programski jezik koji koristimo za izvođenje operacija nad bazom podataka. Međutim, Hibernate nam nudi vlastitu i „moćniju“ verziju SQL-a koja se naziva HQL - Hibernate Query Language.

HQL dizajniran je kao minimalna, objektno-orijentirana, nadogradnja SQL-a namijenjena da zadovolji potrebe Hibernate-a za mapiranjem između klasa i tablica. Također, HQL pruža brojne benefite naspram korištenja čistog SQL-a [18]:

- Koristeći HQL moguće je SQL upite napisati koristeći Java klase umjesto tablica i stupaca. Primjerice, jednostavan upit za dohvaćanje svih podataka iz tablice u SQL-u bi izgledao ovako: `SELECT * FROM 'table_name';` Isti upit u HQL-u izgleda ovako: `FROM TableName;`, ako uzmemo u obzir da je „TableName“ ime klase u Javi. Konkretan primjer HQL-a bit će prikazan u drugom dijelu rada.
- HQL vraća rezultate upita u obliku objekta.
- HQL podržava polimorfne (eng. polymorphic) upite. Polimorfni upiti vraćaju rezultate zajedno sa objektima djecom (eng. child objects).

- HQL pruža brojne dodatne mogućnosti poput paginacije, kao i lakšeg pridruživanja tablica (eng. join). Osim toga, HQL podržava i agregaciju, grupiranje, sortiranje itd.
- HQL ima svojstvo neovisnosti o bazi podataka (eng. database independency). Znamo kako svaki RDBMS koristi pomalo različitu SQL sintaksu. Hibernate pretvara HQL u SQL specifičan za RDBMS koji se koristi u datom trenutku.

3.6 Hibernate tipovi mapiranja

Sada već znamo kako je mapiranje između objekata i tablica glavna značajka Hibernate-a, a u ovom poglavlju vidjet ćemo kako Hibernate izvodi mapiranje između tipova podataka u Javi i onih u tablici.

Hibernate poznaje i Java i SQL tipove podataka. Međutim, tipove podataka koje deklariramo prilikom konfiguracije mapiranja nisu niti Java, niti SQL tipovi podataka. Ti se tipovi nazivaju Hibernate tipovi mapiranja (eng. Hibernate mapping types). Takav tip podatka sadrži informaciju o Java i pripadajućem SQL tipu podatka te na taj način bez problema može prevoditi iz jednog tipa u drugi [14]. Sljedeća tablica prikazuje Hibernate tipove sa pripadajućim Java i SQL tipovima podataka.

Tablica 1. Hibernate, Java i SQL tipovi podataka (Izvor: Vlastita izrada prema [14])

Hibernate tip mapiranja	**Java tip**	**ANSI SQL tip**
integer	int ili java.lang.Integer	INTEGER
long	long ili java.lang.Long	BIGINT
short	short ili java.lang.Short	SMALLINT
float	float ili java.lang.Float	FLOAT
double	double ili java.lang.Double	DOUBLE
character	Java.lang.String	CHAR(1)
string	Java.lang.String	VARCHAR
byte	byte ili java.lang.Byte	TINYINT
boolean	boolean ili java.lang.Boolean	BIT
yes/no	boolean ili java.lang.Boolean	CHAR(1) ili ('Y' or 'N')
true/false	boolean ili java.lang.Boolean	CHAR(1) ili ('T' or 'F')

date	java.util.Date ili java.sql.Date	DATE
time	java.util.Date ili java.sql.Date	TIME
class	java.lang.Class	VARCHAR
Currency	java.util.Currency	VARCHAR

4. Usporedba na temelju primjera

U prošlim poglavljima vidjeli smo što su to JDBC API i Hibernate. Iz njihovog teorijskog opisa već možemo vidjeti neke razlike među njima. U ovom poglavlju te će razlike biti prikazane putem konkretnih primjera. U tim primjerima moći će se vidjeti kako se JDBC API i Hibernate zapravo koriste u praksi. Točnije, komponente JDBC API-ja i Hibernate-a, koje su opisane ranije u radu, ovdje ćemo vidjeti na djelu. Osim toga, u ovom će poglavlju biti uspoređene i mogućnosti jednog i drugog alata. Iz svega toga, na kraju ćemo moći zaključiti u kojim situacijama je bolje koristiti JDBC API, a u kojima Hibernate.

Kod projekta u kojem se nalaze primjeri koji će biti prikazani u ovom dijelu rada dostupan je na sljedećem GitHub repozitoriju: https://github.com/LukaFilipovic99/jdbc-hibernate-demo (primjeri su raspoređeni u tri git grane te su djelo autora ovoga rada). Projekt se sastoji od tri modula: ***jdbc***, ***hibernate*** i ***common***. U *jdbc* modulu nalaze se primjeri korištenja JDBC API-ja. U *hibernate* modulu nalaze se primjeri korištenja Hibernate-a kao i sva konfiguracija vezana za Hibernate, dok se u *common* modulu nalazi klasa sa pomoćnim (eng. utility) metodama koje se koriste u oba ranije navedena modula. Korištena je MySQL baza podataka koja je podignuta lokalno. Oba modula koriste istu bazu sa jednakim entitetima kako bi njihova usporedba bila što točnija. Od vanjskih biblioteka, na projektu su korištene biblioteke Lombok i MySQL konektor. Lombok je biblioteka koja služi smanjenju ponavljajućeg (eng. boilerplate) koda. To radi na način da dodavanjem pojedine anotacije iznad deklaracije klase ona automatski generira dio koda. Primjerice, anotacijama *@Getter* i *@Setter* automatski će se generirati getteri i setteri za tu klasu. MySQL konektor nam je potreban kako bi uopće mogli spojiti Java program s MySQL bazom podataka. U projektu je korišten i Maven kao alat za automatsku izgradnju Java aplikacije (eng. Java building tool).

4.1 Konfiguracija i spajanje na bazu

U ovom dijelu prvo će se pokazati kako se pomoću ova dva alata Java program uopće povezuje s bazom podataka te kakvu je konfiguraciju potrebno odraditi kako bi se oni mogli koristiti u komunikaciji s bazom.

Najprije treba uvesti (eng. import) već spomenuti MySQL konektor. On je potreban i pri korištenju JDBC API-ja i pri korištenju Hibernate-a. Sljedeći isječak koda prikazuje dio koda u xml formatu koji se treba dodati u Maven-ovu konfiguracijsku datoteku (*pom.xml*) kako bi se uvezao MySQL konektor.

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.0.32</version>
</dependency>
```

JDBC API ne zahtjeva nikakvu dodatnu konfiguraciju te se vrlo lako možemo povezati s bazom podataka i uspostaviti konekciju. Potrebni su nam podaci za pristup bazi:

```
private final static String DATABASE_URL = "jdbc:mysql://localhost:3306/jdbc_hibernate_demo";
private final static String USERNAME = "root";
private final static String PASSWORD = "database99";
```

Zatim, pomoću DriverManager-a vrlo lako možemo uspostaviti konekciju s bazom podataka na sljedeći način:

```
Connection connection = DriverManager.getConnection(DATABASE_URL, USERNAME, PASSWORD);
```

Nakon toga, *connection* objekt možemo koristiti za daljnju komunikaciju s bazom što ćemo vidjeti kasnije u radu.

Što se tiče Hibernate-a, osim uvoženja MySQL konektora, trebamo uvesti i Hibernate zavisnost (eng. dependency) u *pom.xml* datoteku.

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.2.3.Final</version>
</dependency>
```

Ranije u radu vidjeli smo kako su Hibernate-u potrebne dvije konfiguracije kako bi ga mogli koristiti. Prva se odnosi na samu konfiguraciju Hibernate-a gdje također konfiguriramo i spajanje na bazu. Za tu konfiguraciju uglavnom se koristi datoteka pod nazivom *hibernate.cfg.xml* gdje u xml formatu pišemo konfiguraciju Hibernate-a. Konfiguraciju smo mogli napisati i u samoj Javi, ali je za ovaj primjer korištena konfiguraciju putem xml datoteke jer se ona češće koristi. Sadržaj *hibernate.cfg.xml* datoteke vidljiv je u sljedećem isječku koda.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <!-- Database connection -->
        <property name="connection.url">jdbc:mysql://localhost:3306/jdbc_hibernate_demo</property>
        <property name="connection.username">root</property>
        <property name="connection.password">database99</property>

        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.show_sql">true</property>

        <property name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.connection.pool_size">1</property>

        <!-- List of mapping classes -->
        <mapping class="model.Movie" />
    </session-factory>

</hibernate-configuration>
```

U prva tri svojstva (eng. property) postavljamo podatke za pristup bazi slično kao i kod JDBC API-ja. Sljedećim svojstvom *hbm2ddl.auto* možemo konfigurirati da Hibernate automatski kreira tablice u bazi prema klasi u Javi koja predstavlja entitet. Ovdje je to svojstvo postavljeno na *update* što znači da će Hibernate kreirati tablicu ako ona ne postoji ili je ažurirati svaki put kada napravimo neku izmjenu u klasi. Postoje i ostale opcije koje možemo postaviti tom svojstvu kao što su primjerice: *create* kojom kreiramo tablicu, a prijašnja tablica ukoliko postoji se briše, *validate* kojom provjeravamo je li tablica ispravna u odnosu na entitet u Javi bez da se vrše ikakve promjene u bazi, *create-drop* kojom će se tablica izbrisati svaki put kad se zatvori SessionFactory objekt i kreirati ponovo kad se otvori itd. Naravno, možemo mu postaviti vrijednost i na *none* čime Hibernate neće raditi nikakve izmjene, ukoliko ipak želimo sami voditi računa o tablici.

Svojstvom *dialect* zadajemo Hibernate-u SQL dijalekt u koji će prevoditi upite. Kako koristimo MySQL bazu, dijalekt je postavljen na MySQL dijalekt. Svojstvo *hibernate.show_sql* možemo postaviti na *true* ili *false*. Ukoliko ga postavimo na *true* Hibernate će putem logova ispisivati upite nad bazom koje izvršava. Za sad je to svojstvo postavljeno na *false*.
Svojstvo *hibernate.current_session_context_class*, kako piše u Hibernate-ovoj službenoj dokumentaciji [13], brine se o implementaciji sučelja *CurrentSessionContext* koje je zaduženo za praćenje trenutne Hibernate sesije. U ovom slučaju, to je svojstvo postavljeno na *thread* što znači da se o sesiji brine trenutna nit. Svojstvom *hibernate.connection.pool_size* definiramo broj maksimalnih mogućih udruženih konekcija (eng. pooled connections) s bazom. Maksimalan broj koji možemo postaviti je 20, a minimalan 1 kao u ovom slučaju.
U dijelu koda ispod komentara „List of mapping classes“ zadajemo klase u Javi koje predstavljaju entitete (odnosno tablice u bazi) čime Hibernate-u dajemo do znanja koje klase treba mapirati u tablice i obrnuto. Konfiguraciju mapiranja možemo izvesti pomoću JPA anotacija u Javi ili putem xml datoteke. Za ovaj primjer korištene su JPA anotacije. O tome nešto više u sljedećem poglavlju.

Hibernate konfiguraciju potrebno je i učitati u Java program. Za to, kao i za kreiranje *SessionFactory* objekta napravljena je klasa *HibernateUtil* te u njoj metoda *createSessionFactory.* Ta metoda učitava konfiguraciju iz *hibernate.cfg.xml* datoteke te pomoću *Configuration* objekta kreira *SessionFactory* objekt. Sljedeći isječak koda prikazuje *HibernateUtil* klasu.

```java
package util;

import model.Movie;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

import java.util.Objects;

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    public static SessionFactory createSessionFactory() {
        if (Objects.isNull(sessionFactory)) {
            try {
                // Create the SessionFactory from hibernate.cfg.xml
                Configuration configuration = new Configuration();
                configuration.configure("hibernate.cfg.xml");
                configuration.addAnnotatedClass(Movie.class);

                ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
                        .applySettings(configuration.getProperties())
                        .build();

                sessionFactory = configuration.buildSessionFactory(serviceRegistry);
            } catch (Throwable ex) {
                System.err.println("Initial SessionFactory creation failed.");
                ex.printStackTrace();
                throw new ExceptionInInitializerError(ex);
            }
        }
        return sessionFactory;
    }

}
```

4.2 Implementacija entiteta

Za primjer ćemo koristiti entitet *Movie* koji predstavlja film. Slijedi isječak koda koji prikazuje POJO klasu tog entiteta korištenog za primjere s JDBC API-jem.

```
package model;

import lombok.*;

import java.time.LocalDate;

@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Movie {
    private Long id;
    private String title;
    private String directorName;
    private LocalDate releaseDate;

    public Movie(String title, String directorName, LocalDate releaseDate) {
        this.title = title;
        this.directorName = directorName;
        this.releaseDate = releaseDate;
    }
}
```

Anotacije iznad deklaracije klase su anotacije iz već spomenute Lombok biblioteke. Radi jednostavnosti, entitet za sada ima samo sljedeće atribute:

- *id* - tipa *Long* koji predstavlja primarni ključ u tablici
- *title* - tipa *String* koji predstavlja naslov filma
- *directorName* - tipa *String* koji predstavlja ime redatelja filma
- *releaseDate* - tipa *LocalDate* koji predstavlja datum izlaska filma

Koristeći JDBC API, tablicu za opisani entitet trebamo kreirati sami. Kod kojim to možemo napraviti izgleda ovako:

```
public void createTable() {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        Statement statement = connection.createStatement();
        statement.executeUpdate( sql: """
                CREATE TABLE IF NOT EXISTS movies (
                id BIGINT PRIMARY KEY AUTO_INCREMENT,
                title VARCHAR(100) NOT NULL,
                director_name VARCHAR(50) NOT NULL,
                release_date DATE NOT NULL);
                """);

    } catch (SQLException throwables) {
        throwables.printStackTrace();
        System.out.println("Could not connect to the database!");
    }
}
```

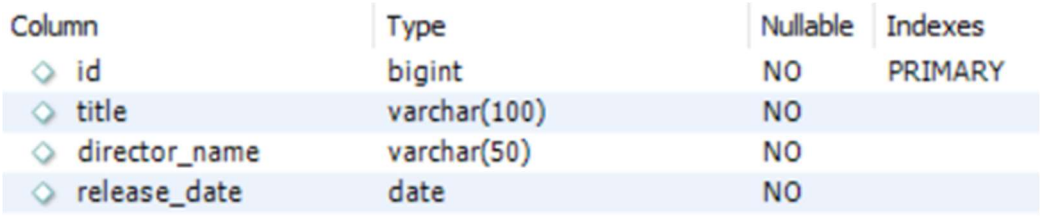
Dakle, nakon što smo stvorili konekciju s bazom, koristimo *Statement* sučelje za izvršavanje SQL koda koji u bazi kreira tablicu *movies*.

U slučaju Hibernate-a, klasa *Movie* izgledat će nam nešto drugačije, a vidljiva je u sljedećem isječku koda.

```java
package model;

import jakarta.persistence.*;
import lombok.*;

import java.time.LocalDate;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString

@Entity
@Table(name = "movies")
public class Movie {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 100)
    private String title;

    @Column(nullable = false, name = "director_name", length = 50)
    private String directorName;

    @Column(nullable = false, name = "release_date")
    private LocalDate releaseDate;

    public Movie(String title, String directorName, LocalDate releaseDate) {
        this.title = title;
        this.directorName = directorName;
        this.releaseDate = releaseDate;
    }
}
```

Ovdje, osim anotacija iz Lombok biblioteke, možemo vidjeti još neke anotacije. To su JPA anotacije kojima konfiguriramo mapiranje između Java klase i tablice u bazi. Anotacijom *@Entity* Hibernate-u dajemo do znanja da je riječ o entitetu, a koristeći anotaciju *@Table* možemo odrediti kako će se u bazi zvati tablica koja predstavlja taj entitet. Anotacijom *@Id* označavamo primarni ključ, a anotacijom *@GeneratedValue* možemo odrediti da nam se id-jevi automatski generiraju. Ukoliko strategiju generiranja postavimo na *GenerationType.IDENTITY,* id-jevi će nam se automatski inkrementirati. Anotacijom *@Column* označavamo retke u tablici. Pomoću te anotacije možemo odrediti i svojstva određenog stupca unutar tablice te odrediti kako će nam se taj stupac zvati u tablici.

Sjetimo se sada svojstva *hbm2ddl.auto* iz konfiguracije Hibernate-a kojeg smo postavili na *update*. To znači da u slučaju Hibernate-a ne moramo ručno kreirati tablicu kao što smo trebali kod JDBC API-ja, već će se to odraditi automatski pokretanjem aplikacije, odnosno kreiranjem *SessionFactory* objekta. Sljedeća slika prikazuje kreiranu tablicu *movies* iz baze u MySQL Workbench alatu. Tablica je ista nakon kreiranja putem JDBC API-ja i nakon kreiranja putem Hibernate-a.



Column	Type	Nullable	Indexes
id	bigint	NO	PRIMARY
title	varchar(100)	NO	
director_name	varchar(50)	NO	
release_date	date	NO	

Slika 9. Tablica *movies* u MySQL Workbench alatu

4.3 Izvođenje operacija nad bazom podataka

U ovom poglavlju bit će prikazano izvođenje osnovnih CRUD operacija putem jednog i drugog alata.

4.3.1 Spremanje

Prva operacija koja će biti prikazana je spremanje jednog zapisa u bazu podataka. Metoda koja to čini koristeći JDBC API može se napisati na sljedeći način.

```
public void save(Movie movie) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        PreparedStatement statement = connection.prepareStatement(
                sql: "INSERT INTO movies (title, director_name, release_date) VALUES (?, ?, ?)");
        statement.setString( parameterIndex: 1, movie.getTitle());
        statement.setString( parameterIndex: 2, movie.getDirectorName());
        statement.setDate( parameterIndex: 3, Date.valueOf(movie.getReleaseDate()));

        statement.executeUpdate();

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

Dakle, ponovo koristimo *Statement* objekt i SQL sintaksu za spremanje u bazu. Ovdje koristimo SQL upit sa parametrima, a te parametre trebamo „zapisati“ u *Statement* objekt i izvršiti naredbu nad bazom.

Metoda koja to isto čini koristeći Hibernate izgleda ovako:

```
public void save(Movie movie) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    // session.save(movie)
    session.persist(movie);

    session.getTransaction().commit();
    session.close();
}
```

Za spremanje koristimo *Session* objekt koji se dobije iz *SessionFactory* objekta i metodu *persist*. U ranijim verzijama Hibernate-a mogla se koristiti i metoda *save* (kao što se vidi u komentaru), ali je ona u verziji Hibernate-a 6 zastarjela (eng. deprecated). Pošto Hibernate „radi s objektima“ dovoljno je samo metodi *persist* predati objekt i Hibernate će sve ostalo odraditi sam kako bi objekt bio spremljen u bazu.

4.3.2 Dohvaćanje

Primjer metode za dohvaćanje svih podataka iz tablice koristeći JDBC API izgleda ovako:

```
public List<Movie> getAll() {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {
        List<Movie> movies = new ArrayList<>();

        PreparedStatement statement = connection.prepareStatement(
                sql: "SELECT * FROM movies");

        ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            Movie movie = new Movie();
            movie.setId(resultSet.getLong( columnIndex: 1));
            movie.setTitle(resultSet.getString( columnIndex: 2));
            movie.setDirectorName(resultSet.getString( columnIndex: 3));
            movie.setReleaseDate(resultSet.getDate( columnIndex: 4).toLocalDate());

            movies.add(movie);
        }

        return movies;
    } catch (SQLException throwables) {
        throwables.printStackTrace();
        return null;
    }
}
```

Ovdje pažnju treba obratiti na *ResultSet* objekt koji čuva podatke dohvaćene iz tablice. Vidimo kako u listu trebamo „ručno“ spremati jedan po jedan *Movie* objekt u koji opet "ručno“ trebamo mapirati podatke iz *ResultSet* objekta. Upravo tu lijepo možemo vidjeti onu glavnu razliku u odnosu na Hibernate koji sam mapira tablice u objekte.

Primjer iste metode u Hibernate-u izgleda ovako:

```
public List<Movie> getAll() {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
    CriteriaQuery<Movie> query = criteriaBuilder.createQuery(Movie.class);
    Root<Movie> rootEntry = query.from(Movie.class);
    CriteriaQuery<Movie> all = query.select(rootEntry);

    TypedQuery<Movie> allQuery = session.createQuery(all);

    List<Movie> movies = allQuery.getResultList();

    session.getTransaction().commit();

    return movies;
}
```

Ovdje je za stvaranje upita korišten *Criteria* objekt. Isto se moglo postići i koristeći HQL, ali primjer s HQL-om bit će prikazan kasnije u radu. Ono što je bitno je metoda *allQuery.getResultList()*. Ta metoda dohvaća podatke iz baze i vraća listu objekata koje onda bez problema možemo spremiti u našu listu koju metoda treba vratiti. Dakle, uopće ne trebamo voditi brigu niti znati na koji način su podaci iz baze prebačeni u listu objekata, jer Hibernate to obavi umjesto nas.

Razmotrimo slučaj u kojem želimo dohvatiti samo jedan *Movie* objekt iz baze, recimo objekt sa određenim id-jem (u pravim full-stack aplikacijama najčešće ćemo dohvaćati objekte prema primarnom ključu). Metodu za to, koristeći JDBC API, možemo napisati na način prikazan sljedećim isječkom koda.

```java
public Movie getById(Long id) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {
        PreparedStatement statement = connection.prepareStatement(
                sql: "SELECT * FROM movies WHERE id=?");

        statement.setLong( parameterIndex: 1, id);
        ResultSet resultSet = statement.executeQuery();

        if (resultSet.next()) {
            Movie movie = new Movie();
            movie.setId(resultSet.getLong( columnIndex: 1));
            movie.setTitle(resultSet.getString( columnIndex: 2));
            movie.setDirectorName(resultSet.getString( columnIndex: 3));
            movie.setReleaseDate(resultSet.getDate( columnIndex: 4).toLocalDate());
            return movie;
        } else {
            return null;
        }

    } catch (SQLException throwables) {
        throwables.printStackTrace();
        return null;
    }
}
```

U Hibernate-u takva metoda izgleda ovako:

```java
public Movie getById(Long id) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    // Movie movie = session.load(Movie.class, id);
    Movie movie = session.get(Movie.class, id);

    session.getTransaction().commit();

    return movie;
}
```

Ovdje opet možemo vidjeti tu glavnu razliku između ova dva alata. Kod JDBC API-ja dohvaćene podatke potrebno je ručno mapirati u objekt, dok nam u Hibernate-u metoda *session.get(Movie.class, id)* automatski vraća željeni *Movie* objekt. U komentaru vidimo i *session.load(Movie.class, id)* metodu koja se može koristiti u starijim verzijama Hibernate-a.

4.3.3 Ažuriranje

Ažuriranje podataka u bazi možemo zamisliti kao nekakvu kombinaciju dohvaćanja prema id-u i spremanja objekta u bazu. Najprije trebamo dohvatiti željeni objekt, zatim mu izmijeniti određene parametre i na kraju ga ponovo spremiti u bazu. Tako to najčešće izgleda kada koristimo Hibernate. Jedina razlika je što se za ponovno spremanje u bazu koristi metoda *session.merge()* umjesto *session.persist()* koja je korištena ranije za spremanje objekta. Sljedeći isječak prikazuje primjer metode koja čini opisano.

```
public void update(Movie movie, Long id) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    Movie existingMovie = session.get(Movie.class, id);
    existingMovie.setTitle(movie.getTitle());
    existingMovie.setDirectorName(movie.getDirectorName());
    existingMovie.setReleaseDate(movie.getReleaseDate());

    session.merge(existingMovie);

    session.getTransaction().commit();
}
```

U JDBC API-ju koristimo SQL izraz za ažuriranje podataka, pa to izgleda ovako:

```
public void update(Movie movie, Long id) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {
        PreparedStatement statement = connection.prepareStatement( sql: """
                UPDATE movies
                SET title = ?, director_name = ?, release_date = ? WHERE id = ?
                """);

        statement.setString( parameterIndex: 1, movie.getTitle());
        statement.setString( parameterIndex: 2, movie.getDirectorName());
        statement.setDate( parameterIndex: 3, Date.valueOf(movie.getReleaseDate()));
        statement.setLong( parameterIndex: 4, id);

        statement.executeUpdate();

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

4.3.4 Brisanje

Brisanje je prilično jednostavno u oba slučaja. Kod JDBC API-ja koristimo SQL sintaksu za brisanje, pa to izgleda ovako:

```
public void delete(Long id) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {
        PreparedStatement statement = connection.prepareStatement(
                sql: "DELETE FROM movies WHERE id=?");

        statement.setLong( parameterIndex: 1, id);
        statement.executeUpdate();

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

U Hibernate-u postoji posebna metoda za brisanje vidljiva u sljedećem isječku koda.

```
public void delete(Movie movie) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    // session.delete(movie);
    session.remove(movie);

    session.getTransaction().commit();
}
```

Kao i za neke ranije metode, metoda *session.remove()* koristi se u novijim verzijama Hibernate-a dok se zastarjela *session.delete()* metoda koristi u starijim verzijama.

4.4 Upravljanje iznimkama

U prethodnim primjerima možemo primijetiti kako su sve metode koje koriste JDBC API zaokružene u try-catch blok. S druge strane, metode koje koriste Hibernate nemaju try-catch blokove. Razlog tome je što metode koje koriste JDBC API izbacuju (eng. throws) *SQLException* što je provjerena iznimka (eng. checked exception). Hibernate, s druge strane, omotava JDBC iznimke te izbacuje *JDBCException* ili *HibernateException* koje su neprovjerene iznimke (eng. unchecked exception). Razlika je u tome što neprovjerenim iznimkama nije potrebno „ručno“ upravljati. U Hibernate-u se za to brine upravljač transakcijama. Zbog toga, nema potrebe pisati ponavljajuće try-catch blokove kao kod JDBC API-ja što uvelike povećava čitljivost koda te ubrzava razvoj aplikacije. [19]

4.5 Veza između tablica

Recimo da našem entitetu koji predstavlja film želimo dodati i žanr. Znamo kako jedan film može imati više žanrova (npr. može biti akcijski i pustolovni). Također, više filmova može pripadati jednom žanru. Dakle potrebna nam je veza više naspram više (eng. many to many, u daljnjem tekstu many to many) između tablica. Pomoću Hibernate-a takvo nešto možemo prilično lako implementirati.

Entitet *Genre* koji predstavlja žanr imat će svoj primarni ključ i ime žanra te će izgledati ovako:

```java
package model;

import jakarta.persistence.*;
import lombok.*;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString

@Entity
@Table(name = "genres")
public class Genre {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 50)
    private String name;

}
```

Sjetimo se, Hibernate će sam kreirati tablicu *genres* u bazi. Samo trebamo paziti da u konfiguracijsku datoteku dodamo put do *Genre* klase na isti način kako je već prikazano i za *Movie*.

Kako bi povezali taj entitet sa entitetom *Movie* odnosno stvorili many to many vezu između te dvije tablice potrebno je dodati sljedeći isječak koda u klasu *Movie*.

```java
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
        name = "movies_genres",
        joinColumns = @JoinColumn(name = "movie_id", nullable = false),
        inverseJoinColumns = @JoinColumn(name = "genre_id", nullable = false)
)
private Set<Genre> genres;
```

Dakle, dodali smo atribut *genres* koji je zapravo set žanrova. Zašto set, a ne lista? Zato što žanr predstavlja entitet, a znamo kako je svaki entitet jedinstven (prema svom primarnom ključu) te isto tako znamo kako set ne može imati duplicirane zapise dok lista može. Zato ovdje koristimo set. Anotacija *@ManyToMany* označava da je riječ o many to many vezi, dok *fetch=FetchType.EAGER* označava da će se objekti djeca (u ovom slučaju *genres*) dohvaćati zajedno sa dohvaćanjem *Movie* objekta. Tu smo mogli još postaviti i *FetchType.LAZY* koji bi označavao da će se objekti djeca dohvatiti samo kada ih mi eksplicitno dohvatimo. Pomoću anotacije *@JoinTable*, kako je prikazano u gornjem isječku koda, Hibernate će kreirati veznu tablicu *movies_genres*, koja nam je potrebna za stvaranje many to many veze, zajedno sa stupcima *movie_id* i *genre_id* unutar te tablice.

Za sljedeći primjer manualno je kreirana tablica „genres“ sa sljedećim podacima:

Tablica 2. Podaci u tablici *genres*

id	**name**
1	Akcijski
2	Horor
3	Pustolovni
4	Komedija

Recimo da sada u tablicu *movies* želimo dodati novi film koji ima žanrove „akcijski“ i „pustolovni“. U Hibernate-u to možemo učiniti na sljedeći način:

```
Genre genreAkcijski = genreDao.getById(1L);
Genre genrePustolovni = genreDao.getById(3L);

Set<Genre> genres = new HashSet<>();
genres.add(genreAkcijski);
genres.add(genrePustolovni);

Movie movie = new Movie(
        title: "The Lord Of The Rings: The Return Of The King",
        directorName: "Peter Jackson",
        LocalDate.of( year: 2003,  month: 12,  dayOfMonth: 1),
        genres);
movieDao.save(movie);
```

Dakle, žanrovi „akcijski“ i „pustolovni“ imaju id-jeve 1 i 3. Njih dohvaćamo metodom *getById* na isti način kako dohvaćamo i *Movie* objekte prema id-u. Te *Genre* objekte koje smo dohvatili spremamo u set te kreiramo novi *Movie* objekt zajedno sa setom žanrova. Zatim možemo koristiti istu onu metodu koju smo i ranije koristili za spremanje *Movie* objekta u bazu. Hibernate će sve ostalo učiniti sam, uključujući i popunjavanje vezne tablice.

Nakon spremanja, ukoliko idemo taj isti film dohvatiti, možemo koristiti istu onu ranije prikazanu metodu za dohvaćanje Movie objekata prema id-u. U ovom slučaju, dohvatit ćemo *Movie* objekt zajedno s *Genre* objektima, odnosno objektima djecom. Ako to ispišemo pomoću *System.out.println* naredbe, uz nadjačanu (eng. override) *toString* metodu, dobit ćemo ispis vidljiv na sljedećoj slici.

```
Fetching movie from database...
Movie{
id=1,
title='The Lord Of The Rings: The Return Of The King',
directorName='Peter Jackson',
releaseDate=2003-12-01,
genres=[Genre(id=3, name=Pustolovni), Genre(id=1, name=Akcijski)]
}
```

Slika 10. Ispis dohvaćenog *Movie* objekta s objektima djecom

Pomoću prikazanih primjera možemo vidjeti kako u Hibernate-u možemo prilično lako i brzo stvarati nove tablice, stvarati veze između tablica te ono najvažnije - za to ne moramo raditi prevelike izmjene u kodu kojeg smo do sad napisali. Da smo ovo isto išli odraditi pomoću JDBC API-ja, osim što bi morali „ručno“ kreirati tablicu *genres* i veznu tablicu *movies_genres*, morali bismo raditi izmjene u metodama koje smo već napisali. JDBC API ne podržava nikakvo povezivanje Java klasa koje predstavljaju entitete, kao što to radi Hibernate. Dakle, primjerice kod spremanja filma u bazu zajedno sa žanrovima (akcijski i pustolovni), morali bismo najprije pomoću jedne metode spremiti film, a onda posebnom metodom spremati id filma i id-jeve žanrova u veznu tablicu. Ukoliko bismo išli dohvatiti taj film zajedno sa žanrovima (objektima djecom) morali bismo raditi izmjene u onoj ranije prikazanoj metodi za dohvat filma prema id-ju. Sljedeći primjeri pokazuju opisani postupak koristeći JDBC API.

Metode za kreiranje tablica *genres* i *movies_genres* izgledaju ovako:

```
public void createTable() {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        Statement statement = connection.createStatement();
        statement.executeUpdate( sql: """
                CREATE TABLE IF NOT EXISTS genres (
                id BIGINT PRIMARY KEY AUTO_INCREMENT,
                name VARCHAR(50) NOT NULL);
                """);

    } catch (SQLException throwables) {
        throwables.printStackTrace();
        System.out.println("Could not connect to the database!");
    }
}
```

```
public void createTable() {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        Statement statement = connection.createStatement();
        statement.executeUpdate( sql: """
                CREATE TABLE IF NOT EXISTS movies_genres (
                movie_id BIGINT NOT NULL,
                genre_id BIGINT NOT NULL,
                PRIMARY KEY (movie_id, genre_id));
                """);

    } catch (SQLException throwables) {
        throwables.printStackTrace();
        System.out.println("Could not connect to the database!");
    }
}
```

Postupak za spremanje novog filma sa žanrovima „Pustolovni“ i „Akcijski“ je isti kao i kod primjera sa Hibernate-om, osim što nakon spremanja filma u bazu, moramo posebno spremiti id-eve u veznu na sljedeći način:

```
movieGenreDao.save(savedMovie.getId(), genreAkcijski.getId());
movieGenreDao.save(savedMovie.getId(), genrePustolovni.getId());
```

Metoda za spremanja id-jeva u veznu tablicu izgleda ovako:

```
public void save(Long movieId, Long genreId) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        PreparedStatement statement = connection.prepareStatement(
                sql: "INSERT INTO movies_genres (movie_id, genre_id) VALUES (?, ?)");
        statement.setLong( parameterIndex: 1, movieId);
        statement.setLong( parameterIndex: 2, genreId);

        statement.executeUpdate();

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

Sada dolazimo do najkompliciranijeg dijela. Ukoliko želimo dohvatiti *Movie* objekt zajedno sa *Genre* objektima djecom potrebno je mijenjati već postojeću metodu za dohvat filma prema id-u. Takva izmijenjena metoda vidljiva je u sljedećem isječku koda.

```java
public Movie getById(Long id) {
    try (Connection connection = DriverManager.getConnection(DATABASE_URL, USERNAME, PASSWORD)) {
        PreparedStatement statement = connection.prepareStatement(
                sql: """
                        SELECT movies.id, movies.title, movies.director_name, movies.release_date,
                        genres.id, genres.name FROM movies
                        JOIN movies_genres ON movies.id = movies_genres.movie_id
                        JOIN genres ON genres.id = movies_genres.genre_id
                        WHERE movies.id=?;
                        """);
        statement.setLong( parameterIndex: 1, id);
        ResultSet resultSet = statement.executeQuery();

        Movie movie = new Movie();
        Set<Genre> genres = new HashSet<>();
        if (resultSet.next()) {
            movie.setId(resultSet.getLong( columnIndex: 1));
            movie.setTitle(resultSet.getString( columnIndex: 2));
            movie.setDirectorName(resultSet.getString( columnIndex: 3));
            movie.setReleaseDate(resultSet.getDate( columnIndex: 4).toLocalDate());

            Genre genre = new Genre();
            genre.setId(resultSet.getLong( columnIndex: 5));
            genre.setName(resultSet.getString( columnIndex: 6));
            genres.add(genre);
        }

        while (resultSet.next()) {
            Genre genre = new Genre();
            genre.setId(resultSet.getLong( columnIndex: 5));
            genre.setName(resultSet.getString( columnIndex: 6));
            genres.add(genre);
        }
        movie.setGenres(genres);
        return movie;

    } catch (SQLException throwables) {
        throwables.printStackTrace();
        return null;
    }
}
```

Prva stvar koja se promijenila je sam SQL izraz. Pošto podatke o žanrovima dohvaćamo iz druge tablice, potrebno je koristiti ključnu riječ *JOIN* za povezivanje tablica. Međutim, ona bitnija promjena vezana je za mapiranje podataka iz ResultSet objekta u Movie objekt. Već je rečeno kako ResultSet vraća podatke u obliku redaka i stupaca. Ono što će nam gornji SQL upit vratiti nalik je sljedećoj tablici.

Tablica 3. ResultSet objekt u obliku tablice

id	title	director_name	release_date	id	name
1	The Lord Of The Rings: The Return Of The King	Peter Jackson	2003-12-01	1	Akcijski
1	The Lord Of The Rings: The Return Of The King	Peter Jackson	2003-12-01	3	Pustolovni

Prva 4 stupca odnose se na podatke o filmu dok se peti i šesti stupac odnose na podatke o žanru. Pošto film ima dva žanra povratni rezultat bit će dva retka sa istim podacima o filmu i različitim podacima o žanru. Na temelju toga, možemo razumjeti način na koji su podaci mapirani u gornjoj metodi. Prvo je bilo potrebno mapirati podatke (iz prvog retka) o filmu u *Movie* objekt i podatke o žanru u *Genre* objekt. Zatim za svaki sljedeći redak mapirati podatke o ostalim žanrovima u nove *Genre* objekte i te objekte spremiti u set. Taj set nakon toga mapiramo u *Movie* objekt. Nakon toga dobit ćemo isti *Movie* objekt (zajedno sa *Genre* objektima djecom) kakav smo dobili i u primjeru s Hibernate-om. Dakle, ono što nam ovdje „komplicira stvari" je to što moramo voditi brigu o izgledu *ResultSet* objekta i smisliti način na koji ćemo to sve mapirati kako bi dobili željeni objekt. To je ono što Hibernate obavi umjesto nas.

Važno je napomenuti kako postoji više načina da se ovakvo nešto izvede pomoću JDBC API-ja. Ovdje je odabran način koji najbolje prikazuje koliko nam uporaba Hibernate-a u ovakvim situacijama može „olakšati život".

4.6 SQL i HQL

U teorijskom dijelu rada opisane su prednosti HQL-a u odnosu na SQL, a ovdje ćemo vidjeti konkretan primjer HQL-a u usporedbi sa SQL-om. Za primjer ćemo uzeti upit koji vraća sve filmove, zajedno s njihovim žanrovima, koji su izašli u određenom vremenskom razdoblju. Takav upit koristeći JDBC API i SQL mogli bismo napisati na sljedeći način:

```
PreparedStatement statement = connection.prepareStatement(
        sql: """
                SELECT movies.id, movies.title, movies.director_name, movies.release_date,
                genres.id, genres.name
                FROM movies
                JOIN movies_genres ON movies.id = movies_genres.movie_id
                JOIN genres ON genres.id = movies_genres.genre_id
                WHERE movies.release_date BETWEEN ? AND ?;
                """);
statement.setDate( parameterIndex: 1, Date.valueOf(startDate));
statement.setDate( parameterIndex: 2, Date.valueOf(endDate));
```

U HQL-u takav upit bi izgledao ovako:

```
Query<Movie> query = session.createQuery( s: """
        FROM Movie as m
        INNER JOIN m.genres as g
        WHERE m.releaseDate BETWEEN :startDate AND :endDate
        """, Movie.class);
query.setParameter( s: "startDate", startDate);
query.setParameter( s: "endDate", endDate);
query.setFirstResult(0);
query.setMaxResults(10);
```

Na ovom primjeru možemo vidjeti prednosti HQL-a opisane u teorijskom dijelu. Prvo, ovaj upit kao rezultat vraća listu filmova koju možemo dohvatiti sljedećom naredbom:

```
List<Movie> movies = query.list();
```

Osim toga, vidimo kako HQL koristi Java klase umjesto imena tablica u svojoj sintaksi. Također, u primjeru se može vidjeti i razlika u povezivanju tablica, odnosno pisanju „join-ova“. U HQL-u smo objekt *Movie* povezali s *Genre* objektima koristeći atribut *genres* iz *Movie* klase, umjesto da smo ih povezivali pomoću id-eva i vezne tablice kao u SQL-u. Treba napomenuti kako HQL ne podržava osnovni *join* već samo: *inner join*, *left outer join*, *right outer join* i *full*

join. Na kraju, vidimo i primjer kako možemo dodati paginaciju. Metodama *setFirstResult* i *setMaxResults* označili smo da želimo dohvatiti samo prvih deset rezultata.

4.7 Performanse

U ovom poglavlju provedena je usporedba performansi JDBC API-ja i Hibernate-a mjerenjem vremena potrebnog za obavljanje određene operacije nad bazom podataka pomoću jednog i pomoću drugog alata. Točnije, uspoređene su performanse prilikom dohvaćanja određenog broja podataka iz baze i spremanja određenog broja podataka u bazu. Vrijeme je mjereno u milisekundama, a treba napomenuti kako su sva mjerenja bila odrađena u istim uvjetima - na istom računalu i nad istom MySQL bazom podataka. Također, dohvaćanje i spremanje izvršeno je samo nad podacima iz tablice *movies*.

Kod dohvaćanja podataka putem JDBC API-ja, mjerilo se i vrijeme potrebno da se rezultati iz *ResultSet* objekta mapiraju u listu s *Movie* objektima kako bi mjerenje bilo što sličnije onome gdje je korišten Hibernate. Također, mjereno je samo vrijeme potrebno za kreiranje upita, izvršavanje upita i mapiranje rezultata. Nije mjereno vrijeme potrebno za stvaranje konekcije s bazom podataka, kao ni učitavanje konfiguracije i kreiranje sesije kod Hibernate-a.

Metoda korištena pri mjerenju performansi JDBC API-ja izgleda ovako:

```
public void measurePerformanceForGet(Integer num) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {
        List<Movie> movies = new ArrayList<>();

        long startTime = System.currentTimeMillis();

        PreparedStatement statement = connection.prepareStatement(
                sql: "SELECT * FROM movies LIMIT ?");
        statement.setInt( parameterIndex: 1, num);

        ResultSet resultSet = statement.executeQuery();

        while (resultSet.next()) {
            Movie movie = new Movie();
            movie.setId(resultSet.getLong( columnIndex: 1));
            movie.setTitle(resultSet.getString( columnIndex: 2));
            movie.setDirectorName(resultSet.getString( columnIndex: 3));
            movie.setReleaseDate(resultSet.getDate( columnIndex: 4).toLocalDate());

            movies.add(movie);
        }

        long endTime = System.currentTimeMillis();

        System.out.println("Number of records: " + num + " | "
                + "Time elapsed: " + (endTime - startTime) + " ms");

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

Sukladno tome, metoda koja čini isto koristeći Hibernate i HQL izgleda ovako:

```
public void measurePerformanceForGet(Integer num) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    long startTime = System.currentTimeMillis();

    Query<Movie> query = session.createQuery( s: "FROM Movie", Movie.class);
    query.setMaxResults(num);

    List<Movie> movies = query.list();

    long endTime = System.currentTimeMillis();

    System.out.println("Number of records: " + movies.size() + " | "
            + "Time elapsed: " + (endTime - startTime) + " ms");

    session.getTransaction().commit();
}
```

Mjerenja su odrađena za 10, 100, 1000, 5000 i 10000 podataka, a rezultati su prikazani u sljedećoj tablici.

Tablica 4. Performanse JDBC API-ja i Hibernate-a prilikom dohvaćanja podataka iz baze

Broj rezultata	JDBC API - vrijeme (ms)	Hibernate - vrijeme (ms)	Omjer
10	48	241	1 : 5.02
100	54	267	1: 4.94
1000	108	315	1: 2.92
5000	175	426	1: 2.43
10000	235	478	1: 2.03

Iz ovih mjerenja možemo zaključiti kako je JDBC API brži prilikom dohvaćanja podataka, a posebno prilikom dohvaćanja manjeg broja podataka. Povećanjem broja dohvaćenih podataka opadaju mu performanse u odnosu na Hibernate radi potrebe za „ručnim“ mapiranjem u objekte. To dokazuje sljedeća tablica s prikazom rezultata koji su dobiveni mjerenjem bez mapiranja rezultata u objekte. Vidljiva je minimalna razlika u vremenima izvođenja u odnosu na broj rezultata. U tom slučaju JDBC API je u prosjeku pet puta brži nego Hibernate.

Tablica 5. Performanse JDBC API-ja prilikom dohvaćanja iz baze bez mapiranja podataka u objekte

Broj rezultata	JDBC API - vrijeme (ms)
10	47
100	48
1000	71
5000	89
10000	110

Što se tiče spremanja podataka u bazu, za mjerenje performansi JDBC API-ja korištena je sljedeća metoda:

```
public void measurePerformanceForSave(Integer num) {
    try (Connection connection = DriverManager.getConnection(
            DATABASE_URL, USERNAME, PASSWORD)) {

        long startTime = System.currentTimeMillis();

        for (int i = 0; i < num; i++) {
            PreparedStatement statement = connection.prepareStatement(
                    sql: "INSERT INTO movies (title, director_name, release_date) VALUES (?, ?, ?)");
            Movie movie = new Movie(
                    CommonUtilClass.generateRandomString( length: 10),
                    CommonUtilClass.generateRandomString( length: 15),
                    CommonUtilClass.generateRandomLocalDate());

            statement.setString( parameterIndex: 1, movie.getTitle());
            statement.setString( parameterIndex: 2, movie.getDirectorName());
            statement.setDate( parameterIndex: 3, Date.valueOf(movie.getReleaseDate()));

            statement.executeUpdate();
        }

        long endTime = System.currentTimeMillis();

        System.out.println("Number of saved records: " + num + " | "
                + "Time elapsed: " + (endTime - startTime) + " ms");

    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}
```

Sukladno tome, metoda za Hibernate izgleda ovako:

```
public void measurePerformanceForSave(Integer num) {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    long startTime = System.currentTimeMillis();

    for (int i = 0; i < num; i++) {
        Movie movie = new Movie(
                CommonUtilClass.generateRandomString( length: 10),
                CommonUtilClass.generateRandomString( length: 15),
                CommonUtilClass.generateRandomLocalDate());

        session.persist(movie);
    }

    long endTime = System.currentTimeMillis();

    System.out.println("Number of records: " + num + " | "
            + "Time elapsed: " + (endTime - startTime) + " ms");

    session.getTransaction().commit();
}
```

Mjerenja su, kao i prilikom dohvaćanja, izvršena na 10, 100, 1000, 5000 i 10000 objekata, a rezultati su prikazani tablicom 6.

Tablica 6. Performanse JDBC API-ja i Hibernate-a prilikom spremanja podataka u bazu

Broj objekata	JDBC API - vrijeme (ms)	Hibernate - vrijeme (ms)	Omjer
10	91	54	1.69 : 1
100	305	191	1.59 : 1
1000	1846	739	2.50 : 1
5000	7472	2256	3.31 : 1
10000	13820	3335	4.14 : 1

Vidimo kako je ovdje situacija obratna u odnosu na dohvaćanje podataka, jer ovdje Hibernate pokazuje bolje performanse. Također, može se vidjeti da se razlika u odnosu na JDBC API povećava kako se povećava broj objekata koje treba spremiti.

4.8 Caching

U ovom poglavlju, najprije treba proučiti sljedeću metodu:

```
public void getByIdCacheExample(Long id) throws InterruptedException {
    Session session = HibernateUtil.createSessionFactory().getCurrentSession();
    session.beginTransaction();

    Movie movie;
    System.out.println("Fetching movie for the first time and printing query...");
    System.out.println("----------------------------------------------------------------------");
    movie = session.get(Movie.class, id);
    System.out.println("Movie: " + movie);

    Thread.sleep( millis: 1000);
    System.out.println("\n");

    System.out.println("Fetching movie for the second time - USING CACHE - no query should be printed");
    System.out.println("----------------------------------------------------------------------");
    movie = session.get(Movie.class, id);
    System.out.println("Movie: " + movie);
}
```

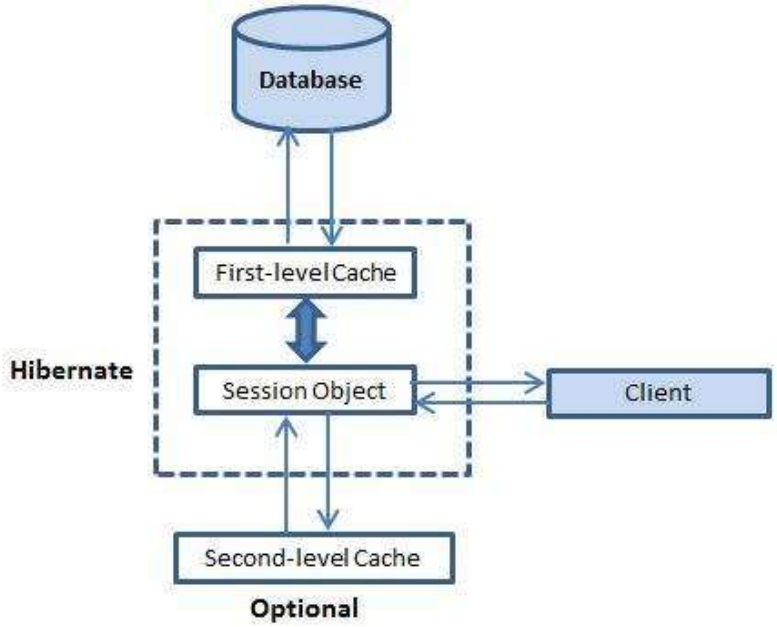
Riječ je o metodi koja, koristeći Hibernate, dva puta dohvaća isti Movie objekt iz baze. Također, u Hibernate-ovoj konfiguracijskoj datoteci, za ovaj primjer, *hibernate.show_sql* svojstvo postavljeno je na *true* kako bi vidjeli upite nad bazom koje Hibernate izvodi. Ispis metode vidljiv je na sljedećoj slici.

```
Fetching movie for the first time and printing query...
----------------------------------------------------------------------
Hibernate: select m1_0.id,m1_0.director_name,m1_0.release_date,m1_0.title from movies m1_0 where m1_0.id=?
Movie: Movie(id=2, title=The Dark Knight: Rises, directorName=Christopher Nolan, releaseDate=2012-07-16)


Fetching movie for the second time - USING CACHE - no query should be printed
----------------------------------------------------------------------
Movie: Movie(id=2, title=The Dark Knight: Rises, directorName=Christopher Nolan, releaseDate=2012-07-16)
```

Slika 11. Ispis *getByIdCacheExample* metode

U ispisu možemo vidjeti kako je prilikom prvog dohvaćanja Hibernate izvršio upit nad bazom. Međutim, prilikom ponovnog dohvaćanja upita nema, a *Movie* objekt je svejedno dohvaćen. Za to je zadužen caching.

Caching je mehanizam kojim se često korišteni podaci spremaju u memoriju kako bi se smanjio broj upita nad bazom. Cilj cachinga je poboljšanje performansi aplikacije. Hibernate podržava **cache prve razine** (eng. first-level cache) i **cache druge razine** (eng. second level cache) što je prikazano slikom 12 [20]. JDBC API, sam po sebi, ne podržava caching mehanizam.



Slika 12. Hibernate cache

(Izvor: https://www.tutorialspoint.com/hibernate/hibernate_caching.htm)

4.8.1 Hibernate cache prve razine

Za ono što smo vidjeli u prethodnom primjeru, zadužen je Hibernate cache prve razine. Takav oblik cache-a inicijalno je uključen u Hibernate-u te se uvijek koristi, a povezan je sa *Session* objektom. Cache prve razine u Hibernate-u ne možemo isključiti, ali postoje metode kojima možemo upravljati njime. Primjerice, metodom *session.evict()* možemo izbrisati određeni objekt iz cache-a, metodom *session.clear()* možemo izbrisati sve objekte iz cache-a, a metodom *session.contains()* možemo provjeriti nalazi li se određeni objekt u cache-u [21].

4.8.2 Hibernate cache druge razine

Za razliku od cache-a prve razine, cache druge razine je opcionalan. Osim toga, vežemo ga za SessionFactory objekt, a ne za Session objekt što znači da ga mogu koristiti sve sesije kreirane istim SessionFactory objektom [22].

Postoje različite implementacije cache-a u Javi koje možemo koristiti kako bi implementirali cache druge razine u Hibernate-u. Najpopularnija među njima je Ehcache. O cache-u druge razine dalo bi se još puno toga napisati kako bi se pokazalo njegovo korištenje u praksi, konfiguracija i slično, ali to nadilazi okvire ovoga rada.

5. Zaključak

JDBC API predstavlja osnovni API za komunikaciju Java aplikacije s bazom podataka. S ciljem da se ubrza i olakša razvoj aplikacija koje imaju potrebu spremati podatke u bazu, razvijeni su ORM frameworci koji pružaju mogućnost automatskog mapiranja Java objekata u tablice u bazi i obrnuto. Najpoznatiji predstavnik ORM frameworka je Hibernate koji se primarno koristi u razvoju web aplikacija.

U radu su kroz pomno osmišljene primjere uspoređeni JDBC API i Hibernate kako bi se vidjele razlike između ova dva alata. Iz usporedbe se može zaključiti kako Hibernate nudi brojne mogućnosti koje JDBC API nema, a koje će najviše doći do izražaja ukoliko radimo sa složenim modelom baze podataka sa višestrukim vezama između tablica. U takvim situacijama, korištenje JDBC API-ja bit će prilično komplicirano, pa je Hibernate svakako bolji izbor. Ipak, ukoliko radimo s jednostavnim modelom baze, u obzir možemo uzeti i korištenje klasičnog JDBC API-ja koji ima puno jednostavniju strukturu te zahtjeva puno jednostavniju konfiguraciju nego Hibernate. Osim toga, JDBC API lakše je shvatiti, pa je tako svakako bolji izbor za početnike u programiranju. U radu je prikazana i usporedba performansi ova dva alata čiji su rezultati pokazali kako JDBC API pruža bolje performanse prilikom dohvaćanja podataka iz baze, dok je Hibernate brži prilikom spremanja podataka u bazu. Također, Hibernate podržava caching mehanizam koji mu služi za poboljšanje performansi, dok JDBC API, sam po sebi, ne podržava caching.

Popis literature

[1] M. Tyson, „What is JDBC? Introduction to Java Database Connectivity“, 2022. [Na internetu]. Dostupno: https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html [pristupano 08.06.2023.]

[2] M. Fisher, J. Ellism J. Bruce, JDBC API Tutorial and Reference, 3. izd., Sun Mycrosystems Inc, 2003.

[3] D. Bales, Java programming with Oracle JDBC, O'Reilly, 2002.

[4] G. Reese, Database programming with JDBC and Java, 2. izd., O'Reilly, 2000.

[5] Tutorialspoint (bez dat.), JDBC - introduction [Na internetu]. Dostupno: https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm [pristupano 11.06.2023.]

[6] Digital Ocean (2022.), CallableStatement in Java example [Na internetu]. Dostupno: https://www.digitalocean.com/community/tutorials/callablestatement-in-java-example [pristupano 10.06.2023.]

[7] Y. Bai, SQL server database programming with Java, Moremedia, Springer, 2022.

[8] Tutorialspoint (bez dat.), Hibernate - ORM overview [Na internetu]. Dostupno: https://www.tutorialspoint.com/hibernate/orm_overview.htm [pristupano 11.06.2023.]

[9] I. V. Abba, „What i san ORM - The Meaning of Object Relational Mapping Database Tools“, 2022. [Na internetu]. Dostupno: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/ [pristupano 11.06.2023.]

[10] TheServerSide (bez dat.), Hibernate [Na internetu]. Dostupno: https://www.theserverside.com/definition/Hibernate [pristupano 11.06.2023.]

[11] GeeksForGeeks.org (bez dat.), Java - Jpa vs Hibernate [Na internetu]. Dostupno: https://www.geeksforgeeks.org/java-jpa-vs-hibernate/ [pristupano 11.06.2023.]

[12] TutorialAndExample.com (2019.), Hibernate History and Versions [Na internetu]. Dostupno: https://www.tutorialandexample.com/hibernate-history [pristupano 11.06.2023.]

[13] Hibernate, službena web-stranica [Na internetu]. Dostupno: https://hibernate.org/ [pristupano 12.06.2023.]

[14] A. A. Puntambekar, Advanced Java (Concepts & Implementation), 1. izd., Technical Publications, 2020.

[15] JavaTPoint (bez dat.), Hibernate Architecture [Na internetu]. Dostupno: https://www.javatpoint.com/hibernate-architecture [pristupano 13.06. 2023.]

[16] Tutorialspoint (bez dat.), Hibernate - Architecture [Na internetu]. Dostupno: https://www.tutorialspoint.com/hibernate/hibernate_architecture.htm [pristupano 11.06.2023.]

[17] Digital Ocean (2022.), Hibernate Criteria Example Tutorial [Na internetu]. Dostupno: https://www.digitalocean.com/community/tutorials/hibernate-criteria-example-tutorial [pristupano 11.06.2023.]

[18] GeeksForGeeks (bez dat.), HQL | Introduction [Na internetu]. Dostupno: https://www.geeksforgeeks.org/hql-introduction/ [pristupano 13.06.2023.]

[19] Digital Ocean (2022.), Hibernate Interview Questions and Answers, [Na internetu]. Dostupno: https://www.digitalocean.com/community/tutorials/hibernate-interview-questions-and-answers [pristupano 27.06.2023.]

[20] Tutorialspoint (bez dat.), Hibernate - caching [Na internetu]. Dostupno: https://www.tutorialspoint.com/hibernate/hibernate_caching.htm [pristupano 27.06.2023.]

[21] Digital Ocean (2022.), Hibernate Caching - First Level Cache [Na internetu], Dostupno: https://www.digitalocean.com/community/tutorials/hibernate-caching-first-level-cache [pristupano 27.06.2023.]

[22] Baeldung (2022.), Hibernate Second-Level Cache [Na internetu]. Dostupno: https://www.baeldung.com/hibernate-second-level-cache [pristupano 27.06.2023.]

COMPARISON OF JDBC API AND HIBERNATE

SUMMARY

This thesis presents comparison of the JDBC API, which is standard API for communication between Java application and database, and Hibernate which is best-known ORM framework. ORM frameworks differ from JDBC API mainly because they provide automatic mapping between Java objects and tables from database. This is also the main advantage of Hibernate over JDBC API. The thesis is devided into two parts. The first part shows theoretical description of JDBC API and Hibernate, their development, architecture and their components, so that the second part of the thesis, where their comparison is presented, would be more understandable. During the comparison, examples were used which are made by the author of this thesis. The comparison was made in order to see the differences in JDBC API and Hibernate configuration, way of connecting to the database, implementation of entities, implementation of relathionships beteween entities, exception handling and way of performing basic CRUD operations. At the end, there is comparison of their performance and description of caching mechanism which is used by Hibernate to improve performance.

Keywords: JDBC API, Hibernate, ORM, database, application, object, table, entity

Popis slika

Popis tablica