JPA

Autori:
Goran Savić
Milan Segedinac

1. JPA

Kao što znamo, u toku izvršavanja aplikacije se podaci čuvaju u objektima u memoriji, dok se dugotrajno skladištenje vrši u bazi podataka. Način organizovanja podataka u objektima nazivamo objektni model, a organizaciju baze podataka, ako koristimo relacionu bazu podataka, nazivamo relacioni model. Ranije smo videli kako se obezbeđuje prevođenje podataka iz jednog modela u drugi. Ovo nazivamo objektno-relaciono mapiranje. Obzirom da postoje prilično jasna pravila za objektno-relaciono mapiranje, ovaj postupak može biti automatizovan.

U ovoj lekciji se bavimo podrškom za automatizovano objektno-relaciono mapiranje u Java aplikacijama. Osnovna ideja pristupa je da se prebacivanje podataka iz objekata u/iz baze podataka delegira posebnoj biblioteci koja će na osnovu konfiguracije objektnog modela preuzeti obavezu komunikacije sa bazom podataka kroz izvršavanje SQL upita. Ovim je programer oslobođen obaveze pisanja SQL upita. Za komunikaciju sa bazom podataka biblioteke koriste JDBC kao bazičnu Java tehnologiju za rad sa bazama podataka.

Java Persistence API (JPA) je zvanična Java specifikacija koja definiše API za objektno-relaciono mapiranje u Java aplikacijama. Postoje različite biblioteke koje implementiraju ovu specifikaciju, a najpopularnija je Hibernate biblioteka, koju i Spring Boot podrazumevano koristi. Biblioteke koje implementiraju JPA specifikaciju se nazivaju JPA provajderi.

Osnovna jedinica organizacije podataka u memoriji je klasa, dok je to u bazi podataka tabela. Prema JPA specifikaciji, za klasu možemo definisati na koju tabelu u bazi podataka se objekti klase mapiraju. To znači da će objekti klase biti uskladišteni kao slogovi specificirane tabele. Podaci u objektu su smešteni u njegovim atributima, dok su u bazi podataka uskladišteni u poljima sloga tabele. Za svaki atribut klase, JPA specifikacija omogućuje specificiranje polja tabele u kojem se podatak iz atributa skladišti. Trenutno standardan način da se definiše objektno-relaciono mapiranje po JPA specifikaciji je korišćenjem Java anotacija. Pogledajmo u sledećem primeru kako možemo specificirati objektno-relaciono mapiranje za objekte koji opisuju državu.

```
@Entity
@Table(name = "tb_country")
public class Country {
	@Id
	@Column(name = "country_id")
	private int id;

	@Column(name = "country_name")
	private String name;

	@Column(name = "country_population")
	private int population;
	...

}
```

Objektno-relaciono mapiranje će biti primenjeno nad objektima klase anotirane anotacijom @Entity. Anotacija @Table određuje u kojoj se tabeli u bazi podataka skladište objekti anotirane klase. Mapiranje atributa na kolone tabele baze podataka vrši se anotacijom @Column u kojoj se navodi naziv polja u tabeli koje odgovara anotiranom atributu. Anotacijom @Id označava se da je primarni ključ tabele polje koje odgovara anotiranom atributu. Pomenimo i da nije neophodno navoditi imena tabele i kolona na

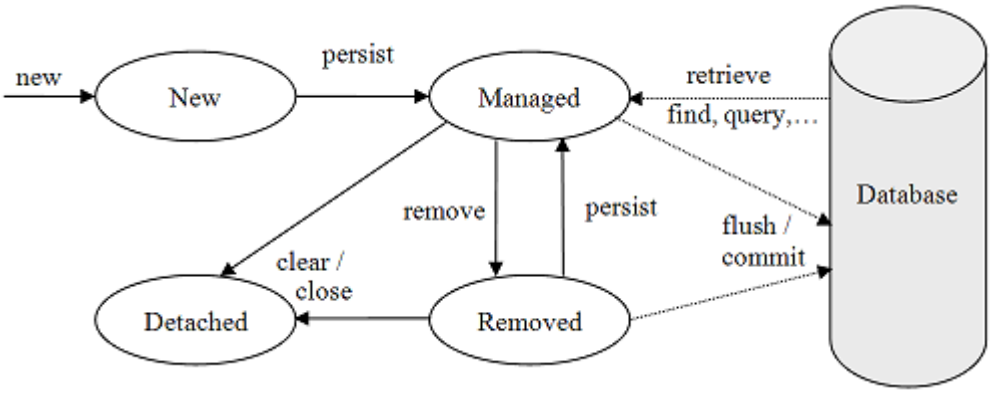
koju se podaci mapiraju, ako se tabela u bazi zove isto kao klasa, odnosno polje u tabeli isto kao atribut klase. Ukoliko za @Table i @Column anotacije nema ni drugih vrednosti, onda se i same anotacije mogu izostaviti.

2. EntityManager

Razmenu podataka sa bazom za objekte ovako anotirane klase preuzeće JPA provajder. Preciznije njegova komponenta pod nazivom EntityManager. Ova komponenta je zadužena za upravljanje entitetima koji se perzistiraju i za razmenu podataka između baze podataka i entiteta u memoriji. Pri obavljanju ovih operacija ova komponenta programski generiše potrebne SQL upite i izvršava ih koristeći JDBC tehnologiju. Pogledajmo na primeru kako EntityManager na sintaksnom nivou realizuje ove operacije, ako se instanca EntityManager klase zove *em* i ako se operacije izvršavaju nad državom predstavljenom promenljivom *country*.

```
Country country = em.find(Country.class, 5); // preuzima državu sa id-om 5
em.persist(country); // snima državu u bazu podataka
em.merge(country); // modifikuje državu u bazi podataka
em.remove(country); // briše državu iz baze podataka
```

Komentari dati pored metoda pojednostavljeno objašnjavaju rezultat rada metoda. Pogledajmo sada detaljnije kako EntityManager upravlja entitetima i šta se zaista dešava pri pozivu metoda iz prethodnog primera. Kolekcija objekata u memoriji kojima EntityManager upravlja naziva se perzistentni kontekts (eng. persistence context). Entitet kojim se upravlja može biti u različitim stanjima koja su predstavljena na narednoj slici.



* preuzeto sa www.objectdb.com/java/jpa/persistence/managed

Tek kreirani objekat nije pod upravljanjem EntityManager objekta i ne nalazi se u perzistentnom kontekstu. Nad ovim objektom je moguće izvršiti operaciju persist koja ubacuje objekat u perzistentni kontekst. Važno je primetiti da smeštanje objekta u perzistentni kontekst nije jednako ubacivanju objekta u bazu podataka. Objekti koji ne postoje u bazi podataka, a nalaze se u perzistentnom kontekstu se ubacuju u bazu podataka pri operaciji *flush* ili *commit*. Operacija *flush* šalje izmene iz perzistentnog konteksta u bazu podataka. Operacija *commit* se izvršava nad transakcijom i predstavlja potvrdu transakcije što uključuje i isporučivanje perzistentnog konteksta u bazu podataka.

Generalno, operacije *flush* i *commit* prebacuju sadržaj perzistentnog konteksta u bazu podataka. Za entitet kojim EntityManager upravlja kažemo da je u stanju *managed*. U ovom stanju će se nalaziti objekti koji su ubačeni u perzistentni kontekst operacijom

persist. Takođe, entiteti dobavljeni iz baze podataka u memoriju (kao u prethodnom primeru metodom *find*) će isto biti u stanju *managed*. Sve promene izvršene nad entitetom u stanju *managed* će EntityManager automatski detektovati i proslediti bazi podataka pri operaciji *flush* ili *commit*.

Uklanjanje entiteta pozivom metode remove EntityManager objekta će samo prebaciti taj entitet u stanje *removed*, a tek će operacija *flush* ili *commit* da brisanje uradi i u bazi podataka. Kao što smo entitet ubacili u perzistentni kontekst, moguće ga je i izbaciti operacijama *clear* ili *close*. Tada se objekat nalazi u stanju detached i EntityManager više njime ne upravlja.

Glavna ideja postojanja perzistentnog konteksta je da se u memoriji čuva samo po jedna instanca entiteta iz baze podataka. Više upita koji preuzimaju iste entitete neće rezultovati kreiranjem više instanci istog entiteta u memoriji. Takođe, ako objekat već postoji u perzistentnom kontekstu, neće biti ponovo dobavljan iz baze podataka, nego će biti preuzet iz konteksta koji je u memoriji. U tom smislu, važna je metoda *equals()* u klasi obzirom da EntityManager utvrđuje da li određeni objekat već postoji u perzistentnom kontekstu time što poredi da li su dva objekta jednaka.

Objasnili smo da metoda *persist* ubacuje u perzistentni kontekst objekat koji se u njemu ne nalazi. Postoji mogućnost i da se u perzistentni kontekst ubaci objekat koji opisuje entitet koji je već u perzistentnom kontekstu predstavljen odgovarajućom instancom. Tada će se izvršiti spajanje podataka iz objekta koji se ubacuje u kontekst sa podacima u objektu koji se već nalazi u kontekstu. To je operacija *merge* i zato se ona koristi pri izmeni entiteta u bazi podataka. Praktično, u kontekst ubacujemo izmenjeni objekat, koji će biti prosleđen bazi podataka pri operaciji *flush* ili *commit*. *Merge* operacija je dozvoljena samo ako predstava entiteta već postoji u kontekstu (npr. entitet je već dobavljen iz baze podataka, pa njegova instanca postoji u kontekstu).

Kao što operacija *flush* šalje podatke iz perzistentnog konteksta u bazu podataka, tako operacija *refresh* osvežava perzistentni kontekst podacima iz baze podataka. Dakle, reč je o inverznoj operaciji koja preuzima podatke iz baze podataka i ubacuje ih u memoriju.

3. Kreiranje šeme baze podataka korišćenjem JPA

Pri mapiranju objektnog na relacioni model, JPA specifikacija omogućuje i specificiranje dodatnih karakteristika tabele i polja koji podatke iz objekata skladište. Ovi podaci se mogu iskoristiti za automatsko kreiranje šeme baze podataka na osnovu anotiranih klasa. Pogledajmo primer sa anotacijom države proširen dodatnim karakteristikama za kolone.

```
@Entity
@Table(name = "tb_country")
public class Country {
	@Id
	@GeneratedValue
	@Column(name = "country_id")
	private int id;



	@Column(name = "country_name", nullable = false, length = 100,
		unique = true)
	@Index
	private String name;
```

```
        @Column(name = "country_population")
        private int population;
        ...

}
```

Vidimo da smo za atribut *name* definisali dodatne karakteristike. Polje u tabeli baze podataka u kojem se ovaj atribut skladišti ne može sadržati vrednost *null*, maksimalni broj karaktera u vrednosti je 255 i naziv države je jedinstven na nivou svih država. Takođe, anotacijom @Index je specificirano da baza podataka treba po ovoj koloni da kreira indeks. Na osnovu ovih vrednosti u anotaciji, automatski će biti generisan kod za skladištenje ovog atributa u koloni tabele baze podataka. Primer kako bi mogao da izgleda generisani kod je dat u nastavku.

```
create table tb_country (
        ...
        name varchar(100) not null unique
        ...
        INDEX ind_name (name)
);
```

Videli smo da se anotacijom @Id označava primarni ključ entiteta. Ranije smo objasnili da je česta praksa da se za primarni ključ bira *auto increment* surogat ključ koji automatski dodeljuje vrednosti primarnog ključa entitetima. Anotacija @GeneratedValue označava da vrednost treba biti automatski određena. Podrazumevano će JPA provajder sam odrediti strategiju kako se vrednosti generišu, ali je moguće i eksplicitno naznačiti strategiju generisanja vrednosti.

4. Veze između objekata

Ranije smo objasnili da se veza asocijacije u objektnom modelu mapira na relacioni model uspostavljanjem spoljnih ključeva. JPA specifikacija pruža anotacije koje opisuju vezu asocijacije između klasa. Formalan opis ovih veza po JPA specifikaciji omogućiće automatske operacije nad povezanim entitetima. Npr. automatsko učitavanje ili brisanje povezanih objekata. Obzirom da veza asocijacije može biti tipa 1:1, 1:N, N:1 i N:N, postoje odgovarajuće anotacije @OneToOne, @OneToMany, @ManyToOne i @ManyToMany.

Pogledajmo na primeru kako možemo opisati vezu asocijacije između države i gradova. Obzirom da jedna država sadrži više gradova, to je veza tipa 1:N gledajući iz perspektive države, a tipa N:1 gledajući iz perspektive grada. Iako postoji način da se definiše i jednosmerna veza asocijacije, mi ćemo prikazati dvosmernu vezu u kojoj država ima informaciju o gradovima koji se u njoj nalaze, a grad informaciju o državi u kojoj se nalazi. Najpre je dat isečak koda koji opisuje ovu asocijaciju u klasi koja predstavlja gradove.

```
public class Place {
        ...
        @ManyToOne
        Country country;
        ...
}
```

Vidimo da grad ima referencu na državu. Obzirom da JPA provajder treba da upravlja ovom vezom asocijacije,ona je anotirana anotacijom @ManyToOne. Naredni isečak koda prikazuje kako je ova veza asocijacije definisana u klasi koja opisuje države.

```
public class Country {
	...
	@OneToMany(mappedBy = "country")
	Set<Place> places = new HashSet<Place>();
	...
}
```

Vidimo da država ima kolekciju gradova, a da anotacija @OneToMany specificira JPA provajderu kako da upravlja ovom vezom asocijacije. Kokretno, vrednost atributa *mappedBy* označava koji atribut u klasi Place predstavlja drugu stranu ove veze asocijacije. Pošto smo u prethodnom primeru atribut koji predstavlja referencu na državu nazvali *country*, onda *mappedBy* ima tu vrednost.

Na sličan način se može uspostaviti asocijacija tipa N:N između entiteta. Vezu ovog tipa ćemo objasniti na primeru veze države sa međunarodnom organizacijom, obzirom da jedna država može biti član više organizacija, a jedna organizacija ima više država članica. Dati su delovi koda klasa koje opisuju državu i međunarodnu organizaciju.

```
public class Country {
	...
	@ManyToMany
	Set<Organization> organizations = new HashSet<Organization>();
	...
}

public class Organization {
	...
	@ManyToMany(mappedBy = "organizations")
	Set<Country> countries = new HashSet<Country>();
	...
}
```

Vidimo da je slično vezi asocijacije 1:N neophodno u jednoj od strana asocijacije definisati atribut *mappedBy* u anotaciji @ManyToMany. Vrednost ovog atributa specificira naziv kolekcije u klasi koja predstavlja drugu stranu ove veze asocijacije.

Primetimo da je u prethodnim primerima izabran skup (klasa Set) za reprezentaciju kolekcija. U skladu sa prirodom perzistentnog konteksta koji za svaki entitet čuva tačno jednu instancu, skup je odgovarajuća kolekcija obzirom da ne skladišti duplikate. Još jedan važan razlog za korišćenje skupa je to što Hibernate biblioteka kao JPA provajder koji se često koristi, ne omogućuje u okviru jednog upita ka bazi povlačenje više listi (kolekcija predstavljenih klasom List). U principu ovo je slučaj kada se radi spoj (eng. *join*) podataka iz više tabela. U narednom poglavlju ćemo analizirati upite ka bazi putem JPA, pa ćemo detaljnije pogledati i ovakav scenario. Ipak, treba naglasiti da ne postoji opšta saglasnost u Java zajednici u pogledu toga koji tip kolekcije koristiti, obzirom da i lista ima svojih prednosti. Na primer, lista je sekvencijalna struktura i poznaje redosled među elementima, dok kod skupa to nije slučaj.

Trenutak preuzimanja povezanih entiteta

Na početku poglavlja smo pomenuli da anotiranje veze asocijacije omogućuje automatizaciju rada sa povezanim entitetima u bazi podataka. Ako država sadrži kolekciju

gradova definisanih JPA OneToMany vezom, pri učitavanju države omogućeno je automatizovano preuzimanje gradova koji se u njoj nalaze. Postavlja se pitanje kada povezani entiteti treba da budu preuzeti iz baze podataka. Ukoliko bi se to učinilo odmah pri preuzimanju glavnog entiteta, postoji opasnost da se tako posredno preuzme veliki broj entiteta, obzirom da i povezani entiteti imaju veze asocijacije sa drugim entitetima. Ovo bi moglo vrlo negativno da utiče na performanse aplikacije. Na primer, učitavanje države bi automatski preuzelo sve organizacije, pa bi se za svaku organizaciju preuzele njene države članice, zatim za svaku državu članicu svi gradovi koji se u njoj nalaze i tako redom.

Iz pomenutog razloga JPA omogućuje programeru da može da kontroliše trenutak kada se povezani objekti preuzimaju iz baze podataka. Preciznije, dve strategije preuzimanja (eng. *fetching*) su omogućene. Prva strategija, pod nazivom *eager*, podrazumeva preuzimanje povezanih entiteta zajedno sa preuzimanjem glavnog entiteta. Druga strategija se naziva *lazy* i rezultuje time da se povezani entitet ili entiteti učitavaju na zahtev, tj. tek kada im se u programskom kodu eksplicitno pristupa. Za primer države koja ima povezanu listu gradova, korišćenje *lazy* strategije bi značilo da će se gradovi učitati iz baze tek pri izvršavanju naredbe *country.getCities()* (ako takva naredba postoji u kodu).

Strategija preuzimanja se zadaje kroz atribut *fetch* anotacija @OneToMany, @ManyToOne i @ManyToMany. Vrednosti su definisane enumeracijom FetchType. Enumeracija sadrži dve moguće vrednosti – EAGER i LAZY. Dat je primer korišćenja ovog atributa.

```
public class Country {
	...
	@OneToMany(mappedBy = "country", fetch = FetchType.LAZY)
	Set<Place> places = new HashSet<Place>();
	...
}
```

```
Country country = em.find(Country.class, 5); // trenutak preuzimanja države
...
// trenutak preuzimanja gradova države, ako je korišćen lazy fetching
List<Place> places = country.getPlaces();
```

Pomenimo i da je kod veze tipa @OneToMany i @ManyToMany podrazumevana strategija LAZY, jer se ove veze odnose na kolekciju entiteta, pa ih je bolje učitavati po potrebi. Sa druge strane, veza tipa @ManyToOne se odnosi na referenciranje jednog entiteta, pa je tu podrazumevana strategija EAGER.

Pri korišćenju *lazy fetching* strategije treba voditi računa da je za naknadno učitavanje povezanih objekata potrebno da u trenutku zahteva za povezanim entitetima postoji aktivan perzistentni kontekst, tj. otvoren EntityManager. Ako perzistentni kontekst više ne postoji ili je glavni entitet izbačen iz njega (stanje *detached*), tada će zahtev za učitavanjem izazvati izuzetak. Ako se koristi Hibernate kao JPA provajder, naziv izuzetka je *LazyInitializationException*.

Kaskadna primena operacija

Osim trenutka preuzimanja iz baze podataka, moguće je definisati i kako se operacija izvršena nad glavnim entitetom odnosi na povezane entitete. Nad entitetom, EntityManager može da izvrši operacije *persist* (dodavanje entiteta u perzistentni kontekst), *merge* (spajanje entiteta sa entitetom koji već postoji u perzistentnom kontekstu), *detach* (izbacivanje entiteta), *remove* (brisanje entiteta), *refresh*

(osvežavanje podataka u entitetu podacima koji su u bazi podataka). JPA omogućuje da se specificira koje od ovih operacija se kaskadno primenjuju nad povezanim entitetima. Na primer, možemo da definišemo da se pri operaciji *persist* izvršenoj nad državom kaskadno izvršava i operacija *persist* nad svim gradovima koji su u tom entitetu država definisani.

Ovo se sintaksno realizuje u atributu *cascade* anotacija @OneToMany, @ManyToOne i @ManyToMany. Pogledajmo prethodni primer sa asocijacijom između države i gradova proširen informacijom o kaskadnom delovanju akcije *persist* na gradove.

```
public class Country {
	...
	@OneToMany(mappedBy = "country", fetch = FetchType.LAZY,
		   cascade = CascadeType.PERSIST)
	Set<Place> places = new HashSet<Place>();
	...
}
```

Vidimo da je primena kaskadne operacije definisana enumeracijom CascadeType. Moguće vrednosti su DETACH, MERGE, PERSIST, REFRESH, REMOVE, kao i ALL, koja označava da se kaskadno delovanje primenjuje na sve tipove operacija.

JPA podrška za vezu nasleđivanja

Ranije smo objasnili da vezu nasleđivanja iz objektnog modela možemo predstaviti u relacionom modelu na tri načina:

- skladištenjem podataka svakog tipa entiteta u posebnoj tabeli
- skladištenjem podataka svih tipova entiteta u jednoj tabeli
- skladištenjem podataka svakog tipa entiteta u posebnoj tabeli, pri čemu tabela postoji samo za one entitete u hijerarhiji koji nisu apstraktni

Obzirom da JPA specifikacija omogućuje automatsko kreiranje šeme baze na osnovu klasa, moguće je specificirati koja od tri pomenute strategije će biti primenjena pri kreiranju šeme. Ovo se definiše postavljanjem anotacije @Inheritance nad klasom koja predstavlja koren hijerarhije nasleđivanja. Tip strategije se definiše atributom *strategy* čije su moguće vrednosti definisane enumeracijom *InheritanceType*. Moguće vrednosti su JOINED, SINGLE_TABLE i TABLE_PER_CLASS, koje redom odgovaraju gore navedenim strategijama mapiranja veze nasleđivanja.

5. JPQL

Važan deo JPA specifikacije je jezik pod nazivom Java Persistence Query Language (JPQL). Ovaj jezik omogućuje preuzimanje objekata iz perzistentnog konteksta, pri čemu se vrši automatsko dobavljanje podataka iz baze podataka u perzistentni kontekst ukoliko je potrebno. Osim operacija *persist, merge, remove i findOne* koje standardno EntityManager podržava, putem JPQL možemo definisati proizvoljan upit za dobavljanje podataka. JPQL je jezik analogan SQL jeziku, s tim što radi sa objektnim modelom (objektima u memoriji) umesto nad relacionim modelom (slogovima u tabelama baze podataka).

Ranije smo videli kako korišćenjem metode *findOne()* EntityManager objekta možemo preuzeti državu sa određenim identifikatorom iz baze podataka. Ako bismo želeli da preuzmemo sve države, neophodno je definisati i izvršiti JPQL upit. Ovo je prikazano u narednom primeru.

```
Query q = em.createQuery("SELECT c FROM Country c");
List<Country> countries = q.getResultList();
```

JPQL upit je predstavljen objektom klase Query. Izvršavanje upita omogućeno je metodom getResultList(). Sam upit se definiše kao string. Pogledajmo strukturu upita iz primera. Upit ima dva dela: SELECT i FROM. SELECT deo označava koji objekti će biti preuzeti kao rezultat upita, a FROM deo definiše odakle će ti objekti biti preuzeti. U konkretnom primeru preuzimaju se objekti klase država kojima je u FROM delu dat naziv *c*. Pored kompletnih objekata moguće je u SELECT delu zatražiti preuzimanje samo određenih podataka iz objekta. Pogledajmo primere ovakvih JPQL upita.

```
Query q1 = em.createQuery("SELECT c.name FROM Country c");
List<String> countryNames = q1.getResultList();

Query q2 = em.createQuery("SELECT c.name, c.population FROM Country c);
List<Object[]> countryData = q2.getResultList();
```

Vidimo da tip rezultata koji upit vraća zavisi od podataka navedenih u SELECT delu upita.

Važno je primetiti da se u upitu navodi ime klase, a ne ime tabele u bazi podataka. Takođe, navode se imena atributa klase, a ne nazivi polja u tabeli. Dakle, JPQL radi nad objektnim modelom. To nam omogućuje da u upitu pristupamo referenciranim objektima do proizvoljne dubine. Pogledajmo primer pristupa nazivu države, ako mesto ima informaciju o državi u kojoj se nalazi.

```
SELECT p.country.name from Place p
```

Filtriranje i sortiranje

Slično SQL-u i JPQL omogućuje filtriranje i sortiranje rezultata. Pogledajmo primer JPQL upita za preuzimanje država koje imaju broj stanovnika veći od 10 miliona i počinju na slovo A sortiranih opadajuće po nazivu.

```
SELECT c FROM Country c WHERE c.population > 10000000 and c.name LIKE 'A%'
ORDER BY c.name DESC
```

Agregatne funkcije i grupisanje

Kao i u SQL-u podržane su i agregatne funkcije COUNT, SUM, AVG, MIN i MAX. Putem GROUP BY klauzule moguće je definisati grupe nad kojima će agregatne funkcije biti primenjene. Pogledajmo primer koji za svako prezime, preuzima koliki broj studenata nosi to prezime.

```
SELECT COUNT(s) FROM Student s GROUP BY s.lastName
```

Izvršavanje ovakvog upita vratiće listu celih brojeva (List<int>).

Preuzimanje podataka iz više entiteta

U prethodnom delu lekcije smo videli da JPA omogućuje definisanje veza asocijacije između objekata. Ova informacija može biti iskorišćena u JPQL upitu za preuzimanje povezanih entiteta. Ovo je analogno korišćenju JOIN klauzule u SQL jeziku, pa se i u JPQL

jeziku koristi isti termin. Pogledajmo primer preuzimanja država koje imaju grad veći od 3 miliona stanovnika.

```
SELECT c FROM Country c JOIN c.places p WHERE p.population > 3000000
```

Dakle, pri spajanju se pristupa kolekciji koja evidentira povezane entitete. Primetimo da JOIN klauzula ovde nema ON deo, zato što je već pri anotiranju entiteta definisano na koji način su povezani država i njeni gradovi (država sadrži kolekciju *places*).

Slično, kao i u SQL jeziku, možemo koristiti LEFT JOIN ako želimo da rezultat sadrži sve objekte levog entiteta bez obzira na to da li ima entitete sa kojima je povezan.

Ne postoji prepreka da se u istom upitu napravi spoj sa više povezanih entiteta. Za ilustraciju, proširićemo prethodni primer time da države moraju biti članice međunarodne organizacije čija je oznaka UN.

```
SELECT c FROM Country c LEFT JOIN c.places p LEFT JOIN c.organizations o
      WHERE p.population > 3000000 and o.code = 'UN'
```

Ako je definisan *lazy fetching* za navedene kolekcije *places* i *organizations*, preuzeti objekti klase *Country* neće imati popunjene ove kolekcije. Možemo u upitu eksplicitno naznačiti da se dobave i ove kolekcije putem reči FETCH, kao u narednom primeru.

```
SELECT c FROM Country c LEFT JOIN FETCH c.places p LEFT JOIN FETCH
c.organizations o WHERE p.population > 3000000 and o.code = 'UN'
```

Sada će preuzeti objekti klase *Country* sadržati i popunjene kolekcije *places* i *organizations*. Napominjemo da će se prethodni upit moći izvršiti samo ako se kao kolekcija koristi skup (klasa Set), a ne lista (klasa List).

Parametri u upitu

Slično JDBC tehnologiji, i JPA omogućuje definisanje upita u kojima su podaci parametrizovani. Pogledajmo primer preuzimanja država sa brojem stanovnika većim od 10 miliona korišćenjem parametrizovanog upita.

```
Query q =
  em.createQuery("SELECT c FROM Country c WHERE c.population > :paramPop");
q.setParameter("paramPop", 10000000);
List<Country> countries = q.getResultList();
```

Vidimo da se parametar u upitu navodi kao *:nazivParametra*. Nakon toga se za parametar postavlja konkretna vrednost.

Kada govorimo o upitima koje JPA specifikacija omogućuje, pomenimo i da je moguće koristiti i SQL jezik. Preporuka je da se klasični SQL upiti koriste samo u slučaju da upit nije moguće realizovati kroz JPQL sintaksu. Pogledajmo primer izvršavanja SQL upita.

```
Query q =
      em.createNativeQuery("SELECT c_name, c_population FROM tb_country");
List<Object[]> countries = q.getResultList();
```

Dakle, koristi se metoda *createNativeQuery* klase *EntityManager*. Treba primetiti da u slučaju klasičnog SQL upita za pristup entitetu koristimo naziv tabele i kolona u bazi podataka.

6. Upravljanje transakcijama putem JPA

Ranije smo videli kako JDBC tehnologija omogućuje programsko upravljanje transakcijama nad bazom podataka iz Java aplikacije. I JPA specifikacija ovo omogućuje, ali na deklarativan način. Naime, nad metodom je moguće definisati anotaciju @Transactional koja definiše da li se i u kojoj transakciji kod naveden u metodi izvršava. Takođe, moguće je anotaciju definisati nad klasom, pa će biti primenjena nad svim metodama klase. Ovo su moguće vrednosti za ovu anotaciju:

- **REQUIRED** – kod metode će se izvršavati u transakciji. Ako metoda nije pozvana iz postojeće transakcije (tj. ako pozivaoc metode nije inicirao transakciju), metoda će kreirati novu transakciju. Ako je metoda pozvana iz postojeće transakcije, kod će se izvršavati u okviru te postojeće transakcije. Ovo je podrazumevana vrednost za anotaciju @Transactional.
- REQUIRES_NEW – metoda uvek inicira kreiranje nove transakcije, bez obzira da li je već pozivaoc kreirao transakciju. Ako je pozivaoc već inicirao transakciju, ta transakcija će biti suspendovana dok transakcija kreirana od strane metode ne bude završena.
- MANDATORY – metoda mora biti pozvana u okviru već postojeće transakcije. Dakle, pozivaoc metode je dužan da započne transakciju. Metoda neće započinjati novu transakciju nego će kod biti izvršen u okviru te postojeće transakcije.
- NEVER – kod se ne sme izvršavati u okviru transakcije. Ako se metoda pozove iz postojeće transakcije, desiće se izuzetak.
- NOT_SUPPORTED – metoda neće sama inicirati kreiranje transakcije. Ako se metoda pozove iz postojeće transakcije, ta transakcija će biti zaustavljena i nastavljena tek nakon što metoda završi rad.
- SUPPORTS – metoda će se izvršiti u okviru postojeće transakcije, ako takva transakcija postoji. Ako ne postoji postojeća transakcija, metoda neće inicirati kreiranje transakcije.

Dat je primer postavljanja anotacije @Transactional.

```
@Transactional(value = Transactional.TxType.REQUIRES_NEW)
public void cancelPayment(int id) {
...
}
```

7. JPA u Spring Boot aplikaciji

Za korišćenje JPA u Spring Boot aplikaciji potrebno je preuzeti biblioteku *org.springframework.boot. spring-boot-starter-data-jpa*.

Osim postavljanja JPA anotacija, u konfiguraciji aplikacije je potrebno podesiti parametre konekcije ka bazi podataka. Ovo se vrši u fajlu `src/main/resources/application.properties`. Dat je primer podešavanja parametara konekcije u ovom fajlu kroz definisanje adrese i porta servera, korisničkog imena i lozinke.

```
spring.datasource.url=jdbc:mysql://192.168.0.2:3306/dbcountry
spring.datasource.username=root
spring.datasource.password=root
```

Osim prikazanih parametara, u ovom fajlu je moguće i dodatno konfigurisati JPA provajder. Pokazali smo da JPA provajder može automatski da kreira šemu baze. Ako je JPA provajder Hibernate, ovo se podešava kroz *property* spring.jpa.hibernate.ddl-auto. Ako se za ovaj *property* postavi vrednost *create,* pri svakom pokretanju aplikacije šema baze će ponovo biti kreirana brišući postojeće podatke. Druga varijanta je vrednost *update* koja vrši modifikaciju šeme baze podataka u skladu sa izgledom objektnog modela ne brišući deo šeme u kojem nije bilo izmena od poslednjeg pokretanja.

Pored pomenutog, moguće je konfigurisati i čitav niz drugih parametara JPA provajdera, kao što su skup karaktera (eng. *character set*) koji se koriste za predstavu podataka ili grupno izvršavanje sličnih upita zbog poboljšanja performansi.

Objasnili smo da se komunikacija sa bazom podataka po JPA specifikaciji vrši korišćenjem EntityManager objekta. U Spring aplikaciji možemo Spring kontejner zadužiti za kreiranje i dobavljanje ovog objekta putem injekcije zavisnosti. Dovoljno je u klasi deklarisati objekat klase EntityManager i anotirati ga anotacijom @PersistenceContext. Dat je primer injektovanja i korišćenja EntityManager objekta u Spring Boot aplikaciji.

```
public class CountryRepository {

	@PersistenceContext
	EntityManager em;

	public List<Country> findAll() {
		Query q = em.createQuery("SELECT c FROM Country c");

		return q.getResultList();
	}
	...
}
```