Spring Data JPA

Autori:
Goran Savić
Milan Segedinac

1. Spring Data JPA

Kao što smo videli, klasičan JPA pristup koristi EntityManager za upite nad bazom podataka. Za svaki entitet se realizuju posebne metode za unos, izmenu, brisanje i peuzimanje. Možemo da primetimo da je za svaki entitet kod u najvećem delu sličan. Kod se može skratiti kroz rad sa generičkim tipom podatka, ali je i dalje potrebno napisati te metode bar jednom. Spring Data JPA je projekat namenjen pojednostavljenu podrške za rad sa bazama podataka u Spring aplikaciji. Osnovna ideja je da se pojednostave klasične Spring tehnike upravljanja podacima kroz kraći kod i ugrađenu podršku za standardne funkcionalnosti. Cilj je da programer u većini situacija ne mora sam da piše upite ka bazi podataka (u SQL ili JPQL sintaksi). U sitaciji kada je neophodno da se upiti pišu, formiranje i izvršavanje upita zaheva značajno manje koda.

Spring Data JPA izbacuje potrebe za pisanjem ponavljajućeg koda tako što uvodi koncept repozitorijuma. Repozitorijum je programska komponenta zadužena za razmenu podataka sa bazom podataka. Repozitorijum je predstavljen kao interfejs, tako da programer ne mora da se bavi implementacijom funkcionalnosti. Postoje različiti tipovi repozitorijuma. Svi tipovi su izvedeni iz pretka *Repository*. Implementacija upravljanja podacima obezbeđuje se kreiranjem interfejsa koji nasleđuje neki od repozitorijumskih interfejsa. Naglašavamo da nije potrebno kreriati klasu koja implementira kreirani interfejs. Ovim se mogu koristiti operacije predviđene nasleđenim repozitorijumskim interfejsom. Pogledajmo koji tipovi repozitorijuma su podržani u Spring Data JPA.

Interfejs *CrudRepository* predstavlja specijalizaciju generičkog repozitorijuma. Ovaj repozitorijum omogućuje standardne CRUD (*create, read, update, delete*) operacije nad entitetom. Za podršku za paginaciju i sortiranje podataka koristi se specijalizacija ovog interfejsa pod nazivom *PagingAndSortingRepository*. Konačno, ovaj interfejs je nasleđen od strane *JPARepository* repozitorijuma. *JPARepository* sadrži dodatnu podrška za JPA operacije (npr. pražnjenje perzistentnog konteksta). Ovaj interfejs pruža sve najčešće operacije potrebne za rad sa podacima u standardnom veb informacionom sistemu Pogledajmo najznačanije metode koje ovaj interfejs sadrži.

```
public interface JpaRepository<T, ID extends Serializable>
       extends PagingAndSortingRepository<T, ID> {

   <S extends T> S save(S entity);      // snimanje entiteta
   T findOne(ID primaryKey);            // preuzimanje entiteta po ključu
   Iterable<T> findAll();               // preuzimanje svih entiteta
   Long count();                        // ukupan broj entiteta
   void delete(T entity);               // brisanje entiteta
   boolean exists(ID primaryKey);       // provera da li entitet postoji
   Page<T> findAll(Pageable pageable);  // preuzimanje jedne stranice entiteta
   Iterable<T> findAll(Sort sort);      // preuzimanje sortiranih entiteta
   ...
}
```

Dovoljno je naslediti prikazani interfejs, da bi u aplikaciji bile dostupne sve funkcionalnosti koje interfejs specificira. Dat je primer nasleđivanja ovog interfejsa.

```
public interface CountryRepository extends JpaRepository<Country, Long> { }
```

Ovim su u aplikaciji dostupne gore navedene operacije za rad sa državama.

2. Query metode

Videli smo kako je korišćenjem Spring Data JPA vrlo jednostavno obezbediti standardne operacije nad entitetom. Što se tiče podrške za nestandardne operacije, i ovo je u Spring Data JPA značajno pojednostavljeno u odnosu na klasičan pristup. U kreirani interfejs se mogu dodavati i nove metode koje vrše operacije nad podacima. Ove metode se ne moraju implementirati, već je dovoljno u interfejsu samo napisati deklaraciju metode.

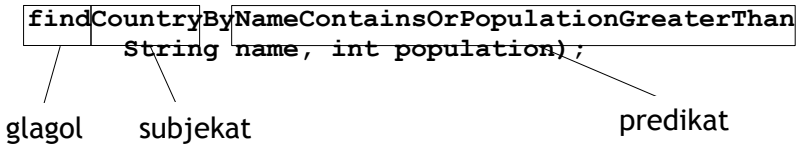
Da bi za definisanu deklaraciju metode Spring Data JPA automatski obezbedio funkcionalnost, metoda mora biti deklarisana poštujući određene konvencije.Ovakve metode se u Spring Data JPA nazivaju *query* metode. Pogledajmo primere *query* metoda za rad sa državama.

```
public interface CountryRepository extends JpaRepository<Country, Long> {
   // preuzima drzave sa naznacenom populacijom
   public List<Country> findByPopulation(int population);

   // preuzima drzave ciji naziv sadrzi prosledjenu rec
   public List<Country> findByNameContains(String name);

   // preuzima drzave ciji naziv sadrzi odredjenu rec ili imaju
   // populaciju vecu od zadate
   public List<Country> findByNameContainsOrPopulationGreaterThan(
      String name, int population);
}
```

Vidimo da nijedna metoda nema implementaciju, već samo deklaracija metode prati određenu konvenciju. Analizirajmo strukturu deklaracije *query* metode.



Glagol ukazuje na tip operacije. Podržani glagoli su *get*, *read*, *find* i *count*. Pri tome, *get*, *read* i *find* su sinonimi i koriste se kod metoda koje preuzimaju entitete iz baze. Svejedno je koji od ova tri sinonima će biti korišćen. Različiti nazivi postoje samo da bi programer mogao da ima određenu slobodu u nazivu metode. Glagol *count* se koristi kod metoda koje vraćaju broj entiteta.

Subjekat ukazuje na tip entiteta koji se preuzima. Postoji samo zbog čitljivosti i slobode u imenovanju i može se izostaviti, jer Spring taj deo u nazivu metode ignoriše. Naime, tip entiteta koji će biti dobavljen zavisi od parametra koji je prosleđen pri nasleđivanju repozitorijuma, a ne od subjekta *query* metode. Jedina situacija u kojoj se subjekat ne ignoriše je ako sadrži reč *Distinct*. Tada subjekat mora da postoji, jer reč *Distinct* ukazuje da treba preuzeti skup entiteta bez duplikata.

Predikat označava kako će podaci biti filtrirani i sortirani. U predikatu se navode nazivi atributa po kojima se filtrira. Između atributa je moguće definisati logičke operatori *And* i *Or*. Atribut naveden u nazivu metode se poredi sa vrednošću odgovarajućeg parametra metode. Pri tome se uzima u obzir redosled parametara, a ne njihov naziv. Što se tiče kriterijuma poređenja prosleđene vrednosti sa specificiranim atributom, podrazumevano se poredi da li je atribut jednak parametru. Pored toga, može se naznačiti operator za

poređenje nakon imena atributa. Postoje različiti operatori za poređenje, npr. Contains, IsLike, IsStartingWith, IsEndingWith, ...

Kao što smo pomenuli, moguće je u *query* metodi definisati i kriterijum sortiranja. Za ovo se koristi ključna reč *OrderBy* iza koje se navode atributi po kojima se sortira i (opciono) smer sortiranja. Dat je primer proširenja metode iz prethodnog primera kriterijumom za sortiranje država opadajuće po populaciji i rastuće po nazivu.

```
findCountryByNameContainsOrPopulationGreaterThanOrderByPopulationDescNameAsc(String name, int population);
```

3. Podrška za paginaciju i sortiranje

U realnim informacionim sistemima količina podataka koji se skladište u bazi podataka može biti veoma velika. Aplikacija mora imati strategiju preuzimanja podataka tako da se ne ugroze performanse. Ovo se standardno rešava preuzimanjem samo dela podataka koji je trenutno potreban. Ove delove nazivamo „stranice“ podataka, a izdeljivanje u stranice paginacija (eng. pagination). Kada govorimo o paginiranom preuzimanju podataka, ono se odnosi kako na preuzimanje podataka iz baze podataka, tako i na preuzimanje podataka sa servera putem mreže (npr. preko REST API-ja).

Spring Data JPA pruža podršku za paginaciju putem *Page* klase. *Page* objekat sadrži jednu stranicu podataka. Pored toga sadrži i dodatne informacije o toj stranici (redni broj i veličinu stranice), kao i podatke koji se odnose na sve uskladištene podatke (npr. ukupan broj uskladištenih entiteta). Repozitorijum *PagingAndSortingRepository* ima podršku za preuzimanje jedne stranice podataka putem *findAll* metode koja prima podatke o stranici koju je potrebno preuzeti. Informacije o stranici su predstavljene objektom koji nasleđuje interfejs Pageable.

Pogledajmo još jednom pomenutu metodu za straničenje i kako ona može biti pozvana.

```
public interface JpaRepository<T, ID extends Serializable>
      extends PagingAndSortingRepository<T, ID> {

   ...
   Page<T> findAll(Pageable pageable);
   ...
}

Pageable page = new PageRequest(2, 10);
Page<Country> countryPage = countryRepository.findAll(page);
```

Vidimo da za kreiranje objekta koji predstavlja stranicu koristimo klasu *PageRequest* koja nasleđuje interfejs *Pageable*. U konstruktoru smo definisali da objekat odnosi na treću stranicu (jer indeksi stranica kreću od 0), a da je veličina stranice 10 entiteta. Poziv metode findAll će vratiti entitete država koji se nalaze na trećoj stranici. Dakle, to su entiteti počevši od 21. do najviše 30. entiteta.

Pri dizajniranju REST API-ja standardno se pruža podrška samo za paginirano preuzimanje entiteta. U ovu svrhu implementira se metoda koja kao parametre u URL-u zahteva očekuje podatke o stranici. Spring pruža podršku za automatsku konverziju parametara URL zahteva u Pageable objekat. Za automatsku konverziju potrebno je informaciju o stranici upisati u parametar *page* (indeksi stranica kreću od nula), a o veličini stranice u parametar *size*. Pogledajmo primer ovakvog URL-a.

```
http://192.168.0.2/api/countries?page=2&size=10
```

Primer metode serverskog REST API-ja koja obrađuje zahtev ka ovom URL-u je dat u nastavku.

```
@RequestMapping(value = "api/countries", method = RequestMethod.GET)
public ResponseEntity<List<Country>> getCountriesPage(Pageable page) {
      Page<Country> countries = countryService.findAll(page);
      ...
}
```

Vidimo da je *Pageable* objekat kreiran od strane Spring kontejnera na osnovu prosleđenih parametara.

Pomenuti *Pageable* interfejs se koristi i za reprezentaciju informacija o kriterijumima sortiranja. Ove informacije se takođe u Spring aplikaciji automatski preuzimaju iz URL-a i to iz parametra *sort*. Pogledajmo primer URL-a koji sadrži i informaciju da je države potrebno preuzeti sortirano opadajuće po imenu.

```
http://192.168.0.2/api/countries?page=2&size=10&sort=name,desc
```

4. Nestandardni upiti

Query metode oslobađaju programera obaveze da piše SQL ili JPQL upite za većinu standardnih upita. Ipak, postoje komplikovaniji upiti koji se ne mogu opisati *query* metodama. Spring Data JPA i za ovaj scenario pruža podršku koja zahteva pisanje manje količine koda nego kod klasičnog JPA pristupa. U repozitorijumu je moguće definisati metodu koja izvršava proizvoljan upit. Pri tome, ni ovde nema potrebe za implementacijom metode, već se metoda anotira anotacijom @Query čija vrednost sadrži upit dat u JPQL sintaksi.

Pogledajmo primer jedne nestandardne metode koja preuzima sume svih uplata klijenata grupisane po mesecima.

```
@Query("SELECT month(p.paymentDate), sum(p.amount) FROM Payment p
        GROUP BY month(p.paymentDate)")
List<Object[]> getPaymentByMonth();
```

U slučaju još komplikovanijih operacija (npr. koje zahtevaju slanje više upita ka bazi podataka), moguće je injektovati *EntityManager* i koristiti klasičan JPA pristup. Praksa je da se u tim situacijama kreira klasa koja nasleđuje repozitorijumski interfejs i u njoj implementira željeni kod. Na ovaj način, u klasi će biti dostupne i standardne operacije koje obezbeđuje repozitorijumski interfejs, kao i dodatne operacije koje je klasa uvela.

5. Data Transfer Objects (DTO)

Kao što znamo, podaci se u veb aplikaciji drugotrajno skladište u serverskoj aplikaciji i privremeno čuvaju u objektima u memoriji. Ovo nazivamo objektni model podataka u aplikaciji. Takođe, REST API pruža i prima podatke organizovane po određenom modelu. Ako koristimo JSON notaciju za serijalizaciju podataka, onda ovaj model određuje strukturu JSON dokumenata koji će biti razmenjivani između klijenta i servera. Postavlja se pitanje da li ova dva modela treba da budu jednaka. Drugim rečima, da li organizacija i sadržaj entiteta za skladištenje treba da bude identična organizaciji i sadržaju entiteta za razmenu sa klijentom.

Ne postoji opšta saglasnost o odgovoru na prethodno pitanje, ali je česta odluka da ova dva modela ne budu jednaka. Dakle, za transfer podataka između klijenata i servera pravi se poseban model podataka. To znači da pored modela entiteta koji se perzistiraju (najčešće ih nazivamo JPA entiteti), pravimo i model entiteta koji će biti razmenjivani između klijenta i servera. Ove entitete nazivamo *Data Transfer Objects* (DTO).

Glavni argument za korišćenje DTO je to da daje više slobode u implementaciji, obzirom da podaci i organizacija podataka koji se šalju klijentu ne moraju biti isti kao podaci i organizacija za skladištenje. Skladištenje je stvar interne implementacije aplikacije, a REST API je javni interfejs funkcionalnosti aplikacije. Nije dobro da ovaj javni interfejs sadrži informacije koje se tiču interne logike aplikacije. Na primer, koji korisnik je izvršio izmenu nad slogom u bazi podataka može biti važna informacija za internu logiku aplikacije, ali ta informacija ne bi smela da bude javno distribuirana u objektu koji REST API vraća. Zato DTO objekat ne bi sadržao tu informaciju.

Jasno je da će u velikoj meri model DTO objekata biti sličan modelu JPA entiteta. To je i glavni argument protiv korišćenja DTO u aplikaciji, obzirom da se moraju implementirati i održavati dva modela podataka koja su u velikoj meri slična.

Pogledajmo primer DTO klase koja predstavlja državu.

```
public class CountryDTO {
	private Long id;
	private String name;
	private int population;

	public CountryDTO() {
	}

	// kreiranje DTO objekta na osnovu JPA entiteta
	public CountryDTO(Country country) {
		this.id = country.getId();
		this.name = country.getName();
		this.population = country.getPopulation();
	}

	... // get i set metode
}
```

DTO klasa treba da sadrži one informacije o entitetu koje klijentu treba da budu dostupne. Pri transferu podataka sa servera ka klijentu i obrnuto, neophodno je vršiti konverziju između DTO entiteta kao modela za transfer podataka i JPA entiteta kao modela za memorijsku reprezentaciju podataka na serveru. Obzirom da često postoji potreba za kreiranjem DTO entiteta na osnovu JPA entiteta, DTO klasa sadrži konstruktor koji vrši ovu operaciju.

Pogledajmo prethodni primer sa paginiranim preuzimanjem država proširen podrškom za DTO objekte. Sada je podatke preuzete iz baze podataka potrebno konvertovati u DTO objekte kako bi oni bili prosleđeni klijentu.

```
@RequestMapping(value = "api/countries", method = RequestMethod.GET)
public ResponseEntity<List<CountryDTO>> getCountriesPage(Pageable page) {
	Page<Country> countries = countryService.findAll(page);
	// prebacivanje rezultata u DTO objekte
	List<CountryDTO> retVal = new ArrayList<>();
	for (Country c: countries) {
		retVal.add(new CountryDTO(c));
```

```
        }
        return new ResponseEntity<>(retVal, HttpStatus.OK);
}
```

Na sličan način se DTO objekat preuzet sa klijenta mora konvertovati u JPA objekat da bi server mogao da vrši dalje operacije nad objektom. Pogledajmo ovo na primeru kreiranja nove države. Podaci o novoj državi stigli su u formi DTO objekta, dok je za perzistenciju države od strane servera potrebno ove podatke reprezentovati u formi DTO entiteta.

```
@RequestMapping(value = "api/countries", method = RequestMethod.POST,
            consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<CountryDTO> create(
        @RequestBody CountryDTO countryDTO) {

  // prebacivanje podataka iz dobijenog DTO objekta u JPA objekat
  Country country = new Country();
  country.setName(countryDTO.getName());
  country.setPopulation(countryDTO.getPopulation());

  country = countryService.save(country);

  return new ResponseEntity<>(new CountryDTO(country), HttpStatus.CREATED);
}
```