SQL - poziomy izolacji

* Read Uncommitted - brak jakichkolwiek ograniczeń. widzimy nawet dane zmienione, ale niezakomitowane przez innych.
* Read Committed - transakcje (T) widzą tylko dane zacommitowane przez inne transakcje.

Czeka z commitem dopóki nie skończy się inna T. Nie ostrzega, że pomiędzy T2 odczytem, a T2 commitem był T1 commit - niebezpieczeństwo nadpisania danych

* Repeatable Read - zapewnia powtarzalność wyników w ramach jednej T.

Czeka z T2 commitem dopóki nie skończy się T1. Jeśli T1 wykonała commit, to T2 nie może zrobić commit i w jego przypadku DB rzuci błędem.

* Serializable isolation - Możliwe są tylko odczyty (może odczytywać kilka jednocześnie), ale też do momentu jeśli nie zacznie się zapis (pojawienie się takiego likwiduje możliwość założenia nowej blokady na tabelę, lub commit założonych wcześniej T odczytujących.

H i update loss (przy Read Committed)

* H domyślnie zapisuje wszystkie pola obiektu (nawet te bez zmian). Można to zmienić używając @DynamicUpdate, ale to jest niezalecane (może być szybsza, ale też i wolniejsze
* Zapobiec można :
  + podnosząc poziom izolacji do Repeatable Read
  + Użyć FOR UPDATE
  + Optimistic locking
* FOR UPDATE - nie pozwoli na wykonanie SELECT przez T2, dopóki T1 nie zakończy się
* Optimistic locking - jest kolumna z numerem wersji, która jest podbijana na każdy update. Jednocześnie taki update ma warunek WHERE sprawdzający, czy numer wersji zgadza się z tym z odczytu poprzedzającego UPDATE. H zlicza ilość wykonanych update i jeśli była ona 0 (a tak będzie jeśli ktoś już zmienił dane z bazie, bo WHERE nie pozwoli na update) i rzuci :
  + StaleObjectStateException
  + OptymisticLockException

Należy pamiętać, że tylko

Cache

* Wpierw odpytywany jest L1, potem L2 a na końcu wysyłane query do DB.
* Jeśli w L1 nie ma wyniku, ale w L2 jest, to jest on kopiowany do L1. Wyniki z query są kopiowane do obu L1 i L2
* Cache L1 - domyślnie włączony. Jest integralną częścią obiektu Sesji. Nie jest dostępny dla innych sesji.
* Cache L2 - domyślnie WYŁĄCZONY
  + Włączenie EnCache jako cache L1 w H

<property name="hibernate.cache.provider\_class">

org.hibernate.cache.EhCacheProvider

</property>  
<property name="hibernate.cache.use\_second\_level\_cache">

true

</property>

-----------------------------------------

<ehcache>

<cache name="com.somecompany.someproject.domain.Country" maxElementsInMemory="10000" eternal="false" timeToIdleSeconds="300" timeToLiveSeconds="600" overflowToDisk="true" />

</ehcache>

* query cache - pobranie wyników z query (list(), scroll() etc) pomija L1.
  + włączenie query cache

<property name="hibernate.cache.use\_query\_cache"> true </property>

* + używanie query cache

query = session.createQuery("from Person p where p.id=1"); **query.setCacheable(true);**

it = query.list().iterator();

* Pobieranie Session i SessionFactory w Sspringu
  + Session s = em.unwrap(Session.class); //@Autowired EntityManager em
  + SessionFactory sf = emf.unwrap(SessionFactory.class); //@Autowired EntityManagerFactory emf
* Usuwanie danych z cache L1
  + s.evict(*entity*)*;*
  + s.clear();
* Usuwanie danych z cache L2
  + cache.evict(Person.class, personPK);
  + cache.evict(Person.class);
  + cache.evictAll();
* Podglądanie cache L1
  + session.contains(*entity*) - entity jest sprawdzane po hash, więc inne entity i takim samym id (nawet wierny klon) nie odda true

SessionStatistics ss = s.getStatistics();

Set<EntityKey> sk = ss.getEntityKeys();

Iterator<EntityKey> ite = sk.iterator();

EntityKey k = ite.next();

boolean czyKlasa = k.getEntityName().equals( Test1.class.getName() );

boolean czyIdVal = k.getIdentifier().toString().equals(Long.valueOf(t0.getId()).toString())

* Podglądanie cache L2
  + sf().getCache().containsEntity( Test1.class, t1.getId() )
  + Cache cache = em.getEntityManagerFactory().getCache();  
    String personPK = ...;  
    if (cache.contains(Person.class, personPK)) {...}

Możliwe stany obiektu:

* transient - obiekt nieobecny w bazie i w PC. Prawdopodobnie nowo utworzony obiekt.
* persistent - obiekt obecny w PC (ale niekoniecznie w DB). Podczas *flush* zapewniamy jego istnienie w DB
* detached - obiekt pierwotnie połczączony z PC, ale już nie (z powodu, *evict*, *s.close()* lub deserializacji/serializacji)

@Entity (@E)

* **Bardzo uważać z @Data !** Zamiast tego lepiej stosować @Getter i @Setter.
  + Program potrafi wpaść w pętlę jeśli dwie klasy wzajemnie się do siebie odwołują
  + Występują błędy nieswkazujące na lomboka, np: “Found two representations of ...”
* klasa i pole nie mogą być final
* musi istnieć konstruktor bez argunentowy
* JPA odwołuje się bezpośrednio do pól albo do getX i setX (GS)
* @E może extends nonEntity i na odwrót
* pola muszą być private, protected, package-private o dostęp przez metody, bo JPA może działać przez proxy obiektu i dostęp bezpośredni może wskazywać po chwili na coś innego
* @javax.persistence.Transient spowoduje pominięcie pola przez JPA
* @Version - wskazuje kolumnę do Optimistic Lock.

@org.hibernate.annotations.OptimisticLock(excluded = true) wyłacza podbijanie tego licznika

dla @Many

* Można używać persistent fields i/lub persistent properties i JPA będzie odwoływało się przez field lub GS. Zasadniczo powinno używać się field bo GS będzie generowało zbędny kod, który dodatkowo może okazać się szkodliwy (nie wiemy kiedy będą GS uruchamiane) <https://stackoverflow.com/questions/594597/hibernate-annotations-which-is-better-field-or-property-access>
* Kolekcje: Collection, Set, Map, List
  + key, value: basic, @E, @Embedded (@EB)
  + dla values typu basic lub @EB: @ElementCollection (@EC) (def: nazwaParent\_nazwaPola)
  + dla values values typu @E: @[O,M]/[O,M]
  + mapy używać tylko po jednej stronie bidirectional (ale dla @MM też???)
  + <O> dla key basc, @MapKeyColumn ustala nazwę kolumny (def: pole+”\_key”)
  + <O> dla key @E, @MapKeyJoinColumn() / @MapKeyJoinColumns ([{@MKJC,...}](mailto:%7b@MKJC,...%7d))

|  |
| --- |
| hibernate01.oneManyEmbdedded  @Embedded NIE JEST KONIECZNE  @Embeddable //@Embeddable – nie posiada Id  **public** **class** Address1 {**public** String city;...}  @Embeddable//@Embeddable – nie posiada Id  **public** **class** Feature1 **public** String city;...}  @Entity **public** **class** Feature2 {@Id @GeneratedValue Integer id;…}  @Entity  **public** **class** Person1 {  @Id  @GeneratedValue  **public** Integer id;  **public** String name;  //Address1: @Embeddable – pola zostaną łączone do tabeli Person1  @Embedded **public** Address1 address;  //Feature1: @Embeddable – osobna tabela “person1\_features” + pole "person1\_id"  //@CollectionTable(name = "person\_featureZZ", joinColumns = @JoinColumn(name="person\_idZZ"))  @ElementCollection() **public** List<Feature1> features = **new** ArrayList<Feature1>();  // osobna tabela “person1\_strings” + pole "person1\_id"  @ElementCollection List<String> strings = **new** ArrayList<>();  // Dokładnie j.w.  @ElementCollection(targetClass=String.**class**) **public** List annoStrings = **new** ArrayList();  //Jak lista – “person1\_features” + pole "person1\_id" i pole “features\_key”  @ElementCollection(fetch=FetchType.***EAGER***) **public** Map<Integer, Feature1> features = **new** HashMap<>();  //Feature1: @Entity – DZIAŁA, ALE NIE UŻYWAC!!! Wymaga wpierw zapisu elementów, a potem Person1  @ElementCollection(fetch=FetchType.***EAGER***) **public** Map<Integer, Feature2> features2 = **new** HashMap<>();  // POPRAWNE!!! CascadeType.All – automatyczny zapis, update i usuwanie – bez wymaga wpierw zapisu  @OneToMany(fetch=FetchType.***EAGER***, cascade=CascadeType.***ALL***)  **public** Map<Integer, PersonFeature2> features3 = **new** HashMap<>(); |
| hibernate01.id  @Entity  **public** **class** Person3\_x {  @Id  //OK! Long, Integer, Short, BigInteger, BigDecimal + java.util.UUID  //ŹLE! String, Float  //AUTO – sam dobiera strategię, sekwencja HIBERNATE\_SEQUENCE lub TableGenetator  @GeneratedValue  **public** Long key1;  //Tworzony pole BIGSERIAL. Dla String zapisyje wartość w „”  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **public** Long/String key1;  //Tworzony osobny generator: person3\_2\_key1\_seq  @GeneratedValue(strategy = GenerationType.***SEQUENCE***)  //j.w. – ustalamy customowe wartości – w DB będzie to „sequenceName”  @GeneratedValue(strategy = GenerationType.***SEQUENCE,*** generator="seq33gen")  @SequenceGenerator(name="seq33gen, sequenceName = "seq33DBgen, initialValue = 4)  **public** Long key1;  //Tworzony osobną tabelę, która ma pola id, next\_val  @GeneratedValue(strategy = GenerationType.***TABLE***)  //j.w. – ustalamy customowe wartości  @GeneratedValue(strategy = GenerationType.***TABLE***, generator="tab34gen")  @TableGenerator(="tab34gen",table = "tab34Id", pkColumnName = "id", pkColumnValue = "val42")  **public** Long key1;  //Pola key1 i key2 będą w Person3\_x, a Person4Id to tylko logiczne opakowanie  @Entity  @IdClass(Person4Id.**class**)  **public** **class** Person4\_11 {  //@GeneratedValue – PRZESŁONI działanie @IdClass !!! – nie używać !!!  @Id **public** Long key1;  ...  @Embeddable //JEST KONIECZNE !!!  **public** **class** Person4Id **implements** Serializable {  **public** Integer key1;  **public** String key2;    **public** Person4Id(Integer key1, String key2) {  **super**();  **this**.key1 = key1;  **this**.key2 = key2;  }  ...  //Dokładnie j.w.  @Entity  **public** **class** Person4\_12 {  @EmbeddedId **public** Person4Id key; |
|  |

* One/Many (O/M) jest dalej
* @Embedded (niby) nie jest konieczne (ale dobrze wygląda), ale w rzeczywistości nie doszło do kompilacji
* @Embeddable może zawierać w sobie inne @Embeddable
* @Embeddable (b) może zawierać w sobie @E (a) i relacja jest od a -> b
* @E może być abstract (to czy ma własną tabelę zależy od strategy: TABLE\_PER\_CLASS)
  + służy głównie do logiki i domyślnie nie będzie miało tabeli
  + nie można utworzyć instancji
  + można zrobić query i wtedy query będzie działało na wszystkich klasach dziedziczących (przetestować co to oznacza, czy to union)
* @MappedSuperclass - nie ma tabeli, nie może być query, nie może być częścią relacji
* @E może dziedziczyć po nie @E, ale zawartość ta jest pomijana przez JPA
* @Inheritance - opisuje konstruowanie tabel w przypadku dziedziczenia:
  + SINGLE\_TABLE - zbije wszystkie klasy w jedną tabelę.
    - Pola oznaczone jako notNull będą null w DB, ale JPA zapobiegnie wstawianiu NULL (kicha dla zewnętrznych narzędzi)
    - jest kolumna discriminatorColumn javax.persistence.DiscriminatorType:STRING(domyślnie nazwa klasy), CHAR, INTEGER opisująca klasę końcową
  + TABLE\_PER\_CLASS - każda końcowa klasa we własnej tabeli (nie wszystkie implementacje JPA to obsługują). Odziedziczone pola są trzymane w wielu tabelach
    - kiepsko odwzorowuje polimorfizm i może wymagać UNION
  + JOINED - poszczególne klasy są trzymane w osobnych tabelach
    - Pomijane są klasy bez @E. Reszta (nawet bez nowych pól) są uwzględniane
    - dobrze odwzorowuje polimorfizm, ale wymaga wielu JOIN aby wyciągnąć cały obiekt
* @ID - może być basic lub obiektem klasy
* Klasa Id (KID):
  + publiczna
  + z publicznym konstruktorem i dostępnymi polami
  + hashCode i equals
  + Serializable
* Złożony PK musi:
  + mieć wiele @Id lub pole być klasy KID
  + musi posiadać @IdClass(kid.class)
  + W tym wypadku jest klucz złożony z 2 pól. CustomerId jest trzymany w kolumnie OrderId, OrderId marowany 2 razy: jako PK i jako FK

@IdClass(kid.Class)

@Id @ManyToOne(“orderId”) customerId

@Id itemId

* Można rozpisać jedną klasę do wielu tabel @SecondaryTable (@ST) . Poniżej zostanie utworzona tabela “x” z PK parent\_id i polem “pole”
  + @SecondaryTable(name=”x”, @PrimaryKeyJoinColumn(s) (name=”id, referencedColumn=”parent\_id”)

@Column (table=”x”) String pole;

* @Lob - opisuje pola BLOB i CLOB. Dla takich pół warto utworzyć @ST, bo zazwyczaj są to pliki i obrazy nie zawsze pobierane z bazy.
* Obsługa dat i czasu definiuje obsługę pół z datą i czasem
  + @Temporal(javax.persistence.TemporalType.TIME/DATE/TIMESTAMP)
  + Dla LocalDate/Time mozna użyć konwertera

@Convert (converter = xxx.Class)

class xxx implements AttributeConverter<LocalDateTime/ o java.sql.TimeStamp>

* One/Many (O/M)
  + <http://in.relation.to/2016/09/28/performance-tuning-and-best-practices/>
  + List vs Set
    - Dla unidirectional używać Set zamiast List, chyba że bardzo zależy na kolejności. Dla List wpierw wszystkie są usuwane, a potem dodawane - dodatkowe zapytania.
    - List nie powala na załadowanie wielu list w jednym FETCH, więc dla n list będzie potrzebnych n zapytań
    - @OrderBy(“xxx ASC”) dla listy wywołuje sortowanie wpisów już w pamięci, a nie w SQL
    - @OrderColumn(name = “xxx”) dla listy powoduje powstanie dodatkowej kolumny xxx wartością indexu
  + Nie zawsze warto robić @OM. Może wystarczy @MO? Należy pamiętać o problemach z Lazy dla @OM (wiele zapytań), a zawsze mozna zrobić @MapsId dla @MO i zrobić dodatkowy FETCH w razie potrzeby.
  + @MM bidirectional - obie strony traktowane jako @OM unidirectional - kiepskie
  + @EC to unidirectional, więc najlepiej używać Set, potem List, a na końcu Bag
  + strona wiodąca
    - posiada relację, czyli w SQL na FK
    - strona wiodąca to **PODOBIEKT**! Czyli **Dla Order**/**Item** wiodącą **będzie Item**
    - @OM i @MO będzie to M
    - @OO i @MM bez znaczenia
    - wiodąca używa @JoinColumn[s] i @InverseJoinColumn[s] dla obiektu “obj”
    - iverse używa @...(mappedBy=”obj”)
    - @OO/MO (wiodąca) @MapsId(“nazwa pola z Id w obiekcie inverse”) - powoduje, że dla OO obie strony mają ten sam klucz, ale dla @MO nie wiem co o tym sądzić
  + @OO (po stronie inverse) i @OM musi mieć cascade lub wcześniej zapisać obiekt (w tej samej transakcji?) wpp. Exception
  + @OO - https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/
    - Typowe

@OneToOne(mappedBy = "post", fetch = FetchType.LAZY, optional = false)

private PostDetails details; **//LAZY nie zadziała ! Zawsze będzie EAGER !**

…

@OneToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

* + - Najlepsze - tylko child ma podobiekt. Child i parent współdzielą klucz parenta, więc mając parenta można wywołać po prostu wczytywanie child - działa tak jakby to było Lazy, ale wymaga dodatkowej linii kodu:

@OneToOne(fetch = FetchType.LAZY)

@MapsId

private Post post;

* + - @OOu @JoinColumn(name=...) - ustala nazwę kolumny
    - @OOb Strona z MappedBy jest inverse, bez MB strony będa traktowane jak podwójny unidirectionall
    - @OOb ma zawsze Fetch ustawi samo na EAGER dla strony wiodącej
  + Kod @OM - https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/
    - Typowe U zakłada tabelę pośredniczącą, więc dla każdego el w liście robi 2 inserty

@OneToMany(

cascade = CascadeType.ALL,

orphanRemoval = true

)

private List<PostComment> comments = new ArrayList<>();

* + - Typowe U z JoinColumn - dla każdego el. na liście robi insert, a potem update “post\_id” (niezależnie czy jest to List czy Set)

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)

@JoinColumn(name = "post\_id")

private List<PostComment> comments = new ArrayList<>();

* + - Najlepsze1 U to … B ! i poleganie na dzieciach do propagowania zmian - robi po jednym insert dla każdego el. listy

@OneToMany(

mappedBy = "post",

cascade = CascadeType.ALL,

orphanRemoval = true

)

private List<PostComment> comments = new ArrayList<>();

public void add/removeComment(PostComment comment) {...}

...

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

* + - Najlepsze2 - tylko @MO - sytuacja podobna jak dla @OO - żeby mieć “dzieci” należy użyć dodatkowego polecenia wczytania

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "post\_id")

private Post post;

* + @MM - jest realizowana poprzez tabelę pośredniczącą:
    - @JoinTable(

name:...,

@JoinColumns({@JoinColumn(name=..,referencedColumnName=)}),

@InverseJoinColumns({@JoinColumn(...)})

)

@ManyToMany(mappedBy:[obj])

* + @JoinColumn(“parent\_id”) - należy uważać przy podobiektach, które należą do wielu list (w jednym lub wielu parentach). Wartość zapisywana (czyli @ID wszystkich możliwych parentów) MUSI pochodzić z jednego generatora, wpp będzie dochodziło do pomieszania danych. Może też zrezygnować z @JoinColumn i zastosować domyślną tabele pośredniczącą.

EntityManager

* Może być zarządzany przez:
  + kontener:
    - automatyczna propagacja persistance context (PC), którego nie przeba ręcznie przekazywać. PC zawiera cache, więc operując na różnych PC możemy otrzymywać różne wyniki, bo PC nie zrzuca od razy danych do bazy.
    - @PersistanceContext

EmtityManager em;

* + - EM nie jest thread safe
  + aplikację:
    - przydatne przy operacjach wielowątkowych
    - @PersistanceUnit

EntityManagerFactory emf;

EntityManager em = emf.createEntityManager();

* cykl życia EM
  + nowy - bez reprezentacji w DB
  + managed - jest w DB i jest związany z PC
  + detached - jest w DB, ale nie jest związany z PC
  + removed - jest w DB i jest związany z PC, ale jest ustalony do usunięcia
* PC wymaga aktywnej transakcji.
  + Transakcje dzielimy na logiczne i fizyczne. Logiczne są zarządzane przez aplikację i mogą mieć pod sobą kilka transakcji fizycznych. Logiczna transakcja może trwać dłuższy czas (np użytkownik myśli) i bardzo niedobrze było by trzymać przez ten czas transakcję fizyczną (jest to anti-pattern oraz DB ma określoną ilość connection, których blokowanie powinno być ograniczone i które powinny być zwalniane kiedy to tylko możliwe).
  + W Springu można jedynie dodać @Transaction do metody lub klasy. Metoda musi być public, więc można zrobić wrapper typu public odpalany z private (2 warappery - jeden void, a drugi zwraca wynik)
  + Spring pozwala zarządzać transakcjami (<https://dzone.com/articles/spring-transaction-management>)
    - programowo (ręcznie) - daje duża kontrolę, ale jest problematyczne przy dużej ilości transakcji i rzadko kiedy jest potrzebne
    - automatycznie (@EnableTransactionManagement)
      * @Transactional (isolation=Isolation.READ\_COMMITTED)
      * pozwala springowi przekazywać transakcję do wywoływanej metody, jeśli ta też jest @Transactional
  + @Transactional(propagation = Propagation.REQUIRE/REQUIRES\_NEW)
    - REQUIRE - wymaga transakcji i utworzy nową tylko gdy rodzic jej nie ma
    - REQUIRES\_NEW - wymaga nowej transakcji
    - Spring nie zapewnia prostego flush. Jeśli chcemy mieć taką możliwość to zaleca się zrobić wrappera z REQUIRES\_NEW - budowa analogiczna do powyższych wrapperów. Commit do PC będzie na wyjście z wrappera

Uruchamianie query / Operacje EM:

* Zapytania można zrobić @Repository implementując CrudRepository<obj.class, id.class>
  + DOA dziedziczące po CrudRepository nie potrzebują konstruktora bezargumentowego.
  + DOA dziedziczące po CrudRepository robią własną transakcję - nie potrzeba @Transactional
  + @Query opisuje SQL, countQuery (dla Page) i native t/f
  + Parametry:
    - “?1” - findByName(String lastname)
    - “:id” - findByIdent(@Param(“id”) Long ident
    - “like %?1” - findByName(String lastname)
  + Repo potrafi wygenerować treść z nazwy metody - patrz ”Tworzenie SQL z nazwy”
  + Repo może przyjmować jako Parametr obiekt Sort lub Page (i oddać Page<...>)
  + @Modifying - należy dodać jeśli query ma modyfikować/dodawać/usuwać dane
    - **UWAGA!** Jeśli zrobimy Delete, to EM nie będzie wiedział o ich usunięciu.
* Operacja dla EM, zamiast powyższego repo.
  + find - odnajduje po id
  + persist - wkłada obiekt do PC. Dla cascade dzieje się to automatycznie.
    - Jeśli obiekt jest już w PC, to jego stan zostanie zaktualizowany (UPDATE)
    - Jeśli obiekt nie istnieje w PC(detach lub inna transakcja), ale istnieje w bazie danych, to będzie DUPLICATE KEY
  + remove - usuwa obiekt z PC. Obiekt mu być w PC, więc wcześniej persist, find lub merge
  + refresh - TYLKO dla obiektu w PC. Pobiera stan z DB i nadpisuje wszystkie zmiany.
  + flush - synchronizuje PC z DB
  + merge - uaktualnia DB o dane z obiektu detached. Zwraca jednocześnie uaktualniony obiekt, więc mergedPerson != person. Przydatne jeśli budujemy obiekt “z palca”, (wypełniamy id) i chcemy wgrać zmiany do DB. Jeśli obiektu docelowego nie ma w PC, to go wpierw dogra z bazy.

Person mergedPerson = (Person) session.merge(person);

* + detach - odłacza obiekt od PC. Wykonany na obiekcie w PC ale nie w DB da EXCEPTION
* Operacja dla Session
  + persist - jw
  + save (tylko H) - jak persist, ale jeśli obj został detached, to będzie INSERT zamiast UPDATE
  + merge - jw
  + update - jak merge, ale nie zwraca nowego, tylko uaktualnia person. Jeśli person nie ma w DB, to rzuci PersistenceException
* Używanie save, update, flush jest niepotrzebne, bo na commit zrobi się to automatycznie (o ile obiekt jest w pc, czyli odczytany lub zmergowany z DB)
* @NamedQuery(name=”yyy”, query=”SELECT … WHERE id = :id\_param”) public class Xxx

Xxx res = em.createNamedQuery(“yyy”)

.setParameter(“id\_param”, <val>)

.getSingleResult() / getResultList() - exception gdy dla single jest wiele

* @NamedQueries( {@NamedQuery(...), @NamedQuery(...)} )
* Query q = em.createNativeQuery("SQL", [Test1.class / "Mapping\_1]");

List<Object> o = q.getResultList();

* native można mapować, o ile podano drugi element. (Test1.class oznacza domyślne mapowanie, ale można użyć customowego - patrz poniżej)

Test5 res = (Test5) o.get(0);

* mapowanie customowe -> patrz dalej

Mapowanie customowe

* @SqlResultSetMapping[s] należy umieścić nad dowolnym @Entity

@SqlResultSetMappings( {

* + Mapowanie Entity@SqlResultSetMapping(

name = "Test5Mapping\_1",

entities = {

@EntityResult(entityClass = Test5.class,

fields = {@FieldResult(name = "intVal1", column = "val2"), ...}) }) ,

* + Mapowanie do listy obiektów (prostych lub złożonych, ale bez rzutowania)

@SqlResultSetMapping(

name = "Test5Mapping\_2",

columns={ @ColumnResult(name="intVal1") }) ,

* + Mapowanie za pomocą konstruktora - do obiektów POJO

@SqlResultSetMapping(

name = "Test5Mapping\_3",

classes={

@ConstructorResult(targetClass=Test5\_1.class,

columns={

//Kolejność pól jest BARDZO WAŻNA - jak w konstruktorze

@ColumnResult(name="id", type=Long.class),

@ColumnResult(name="s2", type=String.class) } )})

})

* Query q = em.createNativeQuery("SQL", "Test5Mapping\_3");

List<Test5\_1> o = q.getResultList();

List<Integer> o = q.getResultList();

PQL

* “,” działa jak [INNER] JOIN. Poniższe są równoznaczne (bez sensu, ale to przykład)
  + SELECT p FROM player p, IN(p.teams) t
  + SELECT p FROM player p JOIN p.teams t
* WHERE
  + t.city = ‘xxx’ - string
  + t.league = :lg - obiekt klasy League
  + t.league.sport : sp - obiekt klasy Sport
* FROM person [AS] p, gdzie p jest case insensitive (CIS)
* [INNER ]JOIN, LEFT [OUTER] JOIN
* Poniższe są równoważne i oddadzą p o ile posiada obiekt car

SELECT p FROM Person p JOIN p.car c

* Wyciąganie obiektów z podobiektami w jednym zapytaniu. Poniższe są równoważne i oddadzą p z podobiektem c, O ILE p .car !=null

SELECT p FROM Person p FETCH JOIN p.car

SELECT p, c FROM Person p JOIN p.car c

* literały są w pojedynczych nawiasach
  + ‘duke’
  + ‘’ to escape dla ‘, więc ‘duke’’s’
* Liczby: 12, -1, 12., 12.3, +12.3
* Bolean true/false są CIS
* Parametry
  + :id - nazwany setParamter(String, val)
  + ?1 - pozycyjny setParameter(int, val)
  + pozycyjny musi zaczynać się od 1
  + parametrów używamy tylko w klauzulach WHERE i HAVING
  + nie można mieszać nazwanych i pozycyjnych w jednym zapytaniu
  + nazwane są CS
* BETWEEN
  + tylko dla liczb
  + dla jednej lub obu wartości null da unknown
* LIKE
  + dla null da unknown
  + \_ pojedynczy znak, % 0 lub więcej znaków
  + Poniższe uzna / za znak escape i ‘\_foo’ przejdzie, ale ‘afoo’ już nie

p.name LIKE ‘\\_f%’ ESCAPE ‘\’

* ORDER BY
  + działa tylko z basic lub klasami orderable
  + wartość po której orderujemy musi być zwracana lub być częścią zwracanego obiektu
    - SELECT o FROM Osoba o ORDER BY o.name - OK
    - SELECT o.city FROM Osoba o ORDER BY o.name - ŹLE

JPA w Springu

* Utworzyć bazę (dla PSQL w PGAdmin, bo DBeaver z jakiegoś powodu nie ma uprawnień do pg\_default)
* Dla Java 10 (i pewnie 9) Dodanie build.gradle

compile 'javax.xml.bind:jaxb-api:2.3.0'

* Konfiguracja podłączenia do DB ręczna albo przez properties (tylko do pojedynczej bazy)
  1. ręczna - patrz “Podłączenie springa do wielu baz”
  2. Prze properties - dodać do application.properties poniższe

#Dla PSQL może pojawiać się błąd: Caused by: java.sql.SQLFeatureNotSupportedException: Metoda org.postgresql.jdbc.PgConnection.createClob() nie jest jeszcze obsługiwana. Aby to “naprawić” należy dodać poniższe

spring.jpa.properties.hibernate.temp.use\_jdbc\_metadata\_defaults = false

# Sterownik bazy danych

spring.datasource.driver-class-name=org.postgresql.Driver

# Rodzaj bazy danych

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

# Baza danych

spring.datasource.url=jdbc:postgresql://Ursus/ll/hibernate03

# Username

spring.datasource.username=system

# Password

spring.datasource.password=system

# Czy i jak automatycznie zmieniać strukturę bazy danych

# none - nie wprowadzamy żadnych zmian i nie usuwamy danych

# create - dodaje i usuwa kolumny. Usuwa dane ze wszystkich (nawet tych bez zmian) tabel. Używa CREATE.

# create-drop - jak crop, ale na zakończenie sesji usuwane są tabele, ale tylko te założone przez Hibernate w tej sesji

# update - Tylko dodaje kolumny. Używa ALTER.

# validate - Sprawdza czy struktury DB odpowiadają tym w Entity. DB może mieć coś więcej, ale nie mniej.

spring.jpa.hibernate.ddl-auto=none

spring.jpa.generate-ddl=true

HQL i native w Springu

* Repository

@Repository ()

public interface *NazwaRepo* extends [Crud/JPA]Repository<*NazwaKlasy*, *TypId*>

{

* HQL

@Query("SELECT…WHERE *xxx*.id = :ident")

public *zwracanyTyp(np List<...>)* *NazwaKlasy nazwaMetody*(@Param("ident") int id);

* SQL

@Modifying - wymagane jeśli jest to UPDATE/INSERT

@Query( value =" *NativeQuerySQL* WHERE *xxx*.id = :ident", nativeQuery = true)

public *zwracanyTyp NazwaKlasy nazwaMetody*(@Param("ident") int id);

* @EnableJpaRepositories(basePackages = { "pl.biu" })

Jeśli utworzymy tylko interface, ale bez implementacji, to implementacja zostanie utworzona automatycznie przez mechanizmy springa. W tym celu należy dodać adnotację , która jest zawarta w spring-boot-starter-data-jpa.

* Jeśli chcemy zaimplementować interface, to WSZYSTKIE metody muszą być zaimplementowane (także te odziedziczone po CrudRepository), więc wygodniej zrobić osobne repository bez rozszerzania CrudRepository.

@Component public class *nazwaKlasy*Impl implements *nazwaInterface*{

@PersistenceContext private EntityManager em;

@Override

public *typ nazwaMetody*(...) {

Query q = em.createNativeQuery(“SQL ...:ident”);

q.setParameter("ident", *value*);

@SuppressWarnings("unchecked") //https://stackoverflow.com/a/21487061

List<Integer> r = (List<Integer>)q.getResultList();

return ...

}

}

* Analogicznie można zastosować wywoływanie HQL w innych miejscach programu.
* Tworzenie SQL z nazwy - spring potrafi wygenerować kod SQL z nazwy metody z interfejsu w unicie repozytorium. Wtedy @Query jest zbędna.
  + <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

public List<Test1> findByIntVal1(Integer ntVal1);

* + find
  + First[xxx]/Top[xxx]/
  + [Distinct]*Klasa*
  + By*NazwaPola1*[Or/And]*NazwaPola2*(String *NazwaPola1[*, Integer *NazwaPola2]*);
  + [All]IgnoreCase
  + Asc/Desc
  + List<Person> findByAddress\_ZipCode(ZipCode zipCode) - ‘\_’ to ‘.’ Person.Address.ZipCode.
  + @Query("select u from User u")

Stream<User> List<User>findByEmailAddressAndLastnameAndStream(String emailAddress, String lastname);

...

try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) {  
 stream.forEach(…);  
 }

* Wyniki można zwracać jako stream, ale wywołanie takiego kody musi być pod transakcja

@Query("SELECT a FROM Test1 a")

public Stream<Test1> getAllStream();

* @Async pozwala na asynchroniczne wywołanie zapytania, które, jeszcze nie musiało się skończyć.

@Async

Future<Obj>/CompletableFuture<Obj>/ListenableFuture<Obj> findById(String Id);

* Dziedziczenie/rozszerzanie
  + @NoRepositoryBean wyklucza interface z obsługiwania go przez Springa. Ten interface będzie mógł jednak być rozszerzany
    - Można zrobić generyczne zapytanie

@NoRepositoryBean

public interface Test1GenDao<T, ID extends Serializable> extends CrudRepository<T, ID> {

@Query("SELECT a FROM #{#entityName} a")

public List<T> getAll();

}

* + - Można zrobić tylko “zajawkę”, ale zdefiniować w extends

@NoRepositoryBean

public interface Test1GenDao<T, ID extends Serializable> extends CrudRepository<T, ID> {

public List<T> getAll();

}

…

public interface Test1CustomDao extends Test1GenDao<Test1, Integer>{

@Query("SELECT a FROM Test1 a")

public List<Test1> getAll();

}

* Łączenie interface

Ważne jest aby implementacja miała taką samą nazwę jak interface, ale z dodatkiem “Impl” - choć nie wiem dlaczego. Co istotne niemożliwe jest w takim przypadku istnienie wielu implementacji tego samego repository interface.

public interface CustomDao {

public int getCustomImpl();}

...

@Component

public class CustomDaoImpl implements CustomDao {

@Override

public int getCustomImpl() {...} }

…

public interface TestDao extends CrudRepository<T, Long>, CustomDao {}

* + Jest jeszcze opisane, że jeśli interfejs i implementacja mają @Component("xx") i @Component("xxxImpl"), to beda łączone po tych nazwach, a nie po nazwach klacy i interfejsu, ale to nie działa.

Sortowanie wyników

* <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-six-sorting/>
* Można stosować ręczne tworzenie obiektów, lub Spring Data Web Support, czyli @EnableSpringDataWebSupport. Poniżej opis tylko ręcznego.
* Obiekt sortujący
  + Pole, do którego będziemy się odwoływać zawsze rozpoczyna się od klasy z FROM, czyli dla … FROM Test3 t3 JOIN t3.test2 t2” doda “... order by test3x0\_.id ...”
  + Pola wymienione w sort NIE MUSZĄ być wymienione w bloku SELECT
  + new Sort(Sort.Direction.DESC, "id")
  + new Sort(Sort.Direction.DESC, "id" [, “t2.id”]) - t2.id, bo w notacji HQL będzie to h3.h2.id
* Zapytanie

@Query(value="SELECT t3.intVal1 FROM Test3 t3 JOIN t3.t2 t2 ")

public List<Integer> getSorted(Sort sort);

* Wywołanie

List<Integer> res = *repo*.getSorted(new Sort(...));

Stronicowanie wyników

* <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-seven-pagination/>
* Można stosować ręczne tworzenie obiektów, lub Spring Data Web Support, czyli @EnableSpringDataWebSupport. Poniżej opis tylko ręcznego.
* dodać dependency 'org.springframework.boot:spring-boot-starter-data-rest'
* Tworzenie obiektu pagable, który pozwoli na pobranie 1 strony w rozmiarze 2 elementów, co sprowadza się do "limit 2"
  + Pageable pageable = PageRequest.of(0, 2 [, SORT]);
* SORT - patrz wyżej.
* Tworzenie obiektu pagable, który pozwoli na pobranie 5 strony w rozmiarze 2 elementów (9 i 10 element), co sprowadza się do "limit 2 offset 8"
  + Pageable pageable = PageRequest.of(4, 2);
* W zależności od istnienia transakcji i poziomu izolacji będziemy otrzymywać różne wyniki
  + bez transakcji - kolejne strony będą wczytywały dane po zmianach
  + poniżej REPEATABLE\_READ - kolejne strony wczytają dane po zmianach (I,U,D)
  + powyżej REPEATABLE\_READ - kolejne strony wczytają dane sprzed zmian (I,U,D)
* Zapytanie
  + @Query(value="SELECT t3.intVal1 FROM Test3 t3")

public Page<Integer> getPaged(Pageable pageable);

Zwracać możemy:

* List<> - lista wyników
* Page<> - zawiera w sobie List<> (.getContent()) i informacje o pageable
* Slice<> - Podobne do Page<>, ale pozwala na tworzenie kolejnego/poprzedniego pageable
  + Jeśli w Query występuje FETCH, to trzeba dodać countQuery. W takim przypadku liczba obiektów w Page odnosi się do liczby obiektów Test4, ale **POBRANE ZOSTANĄ WSZYSTKIE WIERSZE** i wynik zostanie obcięty w pamięci.

@Query(value="SELECT t4 FROM Test4 t4 LEFT JOIN FETCH t4.t2",

countQuery="SELECT count(t4) FROM Test4 t4 LEFT JOIN t4.t2")

* + Jeśli w Query występuje FETCH, to countQuery JEST NIEZBĘDNE ! Inaczej występuje błąd “query specified join fetching, but the owner of the fetched association was not present in the select list”

Co prawda opis mówi, że count zostanie utworzony z nazwy metody, ale nie wiem, jak to uzyskać.

* Wywołanie
  + Page<Integer> res = t1d.getPaged(pageable);

List<Integer> lst = res.getContent();

Iterator<Integer> it = res.iterator();

Projekcja wyników

* odpowiednik View w SQL - pozwala zmapować wynik zapytania na **INTERFACE (nie POJO).**
* Można używać z HQL i SQL
* nazwy getterów muszą odpowiadać zwracanym polom w zapytaniu. Można używać “as” jeśłi nazwy są długie
* Kod

public interface SubObj {

public Integer getV3();

public Integer getyyy();

...

@Query(value="SELECT t3.xxx as v3, t3.yyy FROM ...", nativeQuery = true)

public List<SubObj> getSubObj();

Entity graph

* Pozwala na kontrolę Lazy/Fetch podczas wykonywania zapytania
* Składnia

@NamedEntityGraphs ({ [@NamedEntityGraph]+ })

* Płaski entity graph

@NamedEntityGraph(

name = "graph.TestSix.lista",

attributeNodes = {@NamedAttributeNode("lazyList"),..})

* Entity graph z subgraphem

@NamedEntityGraph(

name = "graph.TestSix.lista",

attributeNodes = @NamedAttributeNode(value = "lazyList", subgraph = "inLista"),

subgraphs = @NamedSubgraph(

name = "inLista",

attributeNodes = {@NamedAttributeNode("list"),...}))

* Wywołanie
  + EntityGraph graph = this.em.getEntityGraph("graph.TestSix.lista");

Map hints = new HashMap();

hints.put("javax.persistence.fetchgraph", graph);

Test6 t6 = em.find(Test6.class, Long.valueOf(1), hints);

* + @EntityGraph(value="graph.TestSix.listy", type = EntityGraphType.LOAD)

@Query("SELECT t FROM Test6 t WHERE t.id=?1")

public Optional<Test6> getByIdListyLoad(Long id);

* + @EntityGraph(“lazyList”)

@Query("SELECT t FROM Test6 t WHERE t.id=?1")

* EntityGraphType
  + LOAD - wymienione będą Fetch a pozostała defaultowe
  + FETCH - wymienione będa Fetch a pozostałe Lazy
  + **Jeśli lista jest oryginalnie Fetch a mapowanie ustawi dla niej Lazy, to po wykonaniu zapytania i tak zostanie doczytana** - albo to błąd, albo inna część programu (bądź IDE) doczytuje. Z tego powodu możliwą opcją jest zrobienie wszystkiego Lazy i sterowanie potem za pomocą EntityGraphType.
* H nie potrafi jednocześnie wczytać kilku List i EG nie jest tutaj wyjątkiem. W takim przypadku lepiej używać Set gdy tylko jest to możliwe.

Podłączenie springa do wielu baz

* <https://www.baeldung.com/spring-data-jpa-multiple-databases>
* Skrót
  + Należy utworzyć @Entity w osobnych pakietach
  + Dla każdego połączenia należy utworzyć @Configuration i dodać @EnableJpaRepositories wskazującą na odpowiedni pakiet
    - Ustawiamy dialekt, driver i pakiet do skanowania

@Bean [@Primary]

public LocalContainerEntityManagerFactoryBean userEntityManager() {...}

* + - * em.setJpaPropertyMap(properties); **KONIECZNIE PO** wypełnieniu, bo podłączenie inicjalizuje wartości w em
      * klucze będa inne niż w application np:

spring.jpa.hibernate.ddl-auto -> hibernate.hbm2ddl.auto

* + - Ustawiamy DB, user i pass

@Bean [@Primary]

public DataSource userDataSource() {...}

* + - Ustawiamy transakcję

@Bean [@Primary]

public PlatformTransactionManager [user]TransactionManager() {...}

* @Transactional("[user]TransactionManager") - należy wskazywać na odpowiednią transakcję

Funkcje Postgres

* Są 2 języki sql i plpgsql
  + Poniższe są równoważne i oddają pojedynczy wiersz z 1 komórką z 2 wartościami “(x,y)”

CREATE OR REPLACE FUNCTION public.inctest62(in int, out o1 int, out o2 int)

AS

$$

select $1+1, $1+2;

$$

LANGUAGE sql

* + CREATE OR REPLACE FUNCTION public.inctest61(integer, OUT o1 integer, OUT o2 integer)

RETURNS record

LANGUAGE plpgsql

AS $$

begin

select $1+1, $1+2 into o1, o2;

END;

$$

* Procedury można wywoływać przez EM:
  + Wywoływać **KONIECZNIE w @Transactional**, bo inaczej będzie Exception
    - **createStoredProcedureQuery**(*nazwaSQL*)

StoredProcedureQuery q = em.createStoredProcedureQuery("p61");

q.registerStoredProcedureParameter(1, Integer.class, ParameterMode.IN);

q.setParameter(1, 6);

Integer res = (Integer) q.getSingleResult()

lub

Integer res = q.getOutputParameterValue("o")

* + - **createNamedStoredProcedureQuery**(*nazwaNamed*)
      * wymaga zdefiniowania nad obiektem Entity

StoredProcedureQuery q = em.createNamedStoredProcedureQuery("t61");

q.setParameter("i", 5);

Integer res = (Integer) q.getSingleResult()

lub

Integer res = q.getOutputParameterValue("o")

* + Wywoływać **MOŻNA BEZ @Transactional**
    - **createNativeQuery**("Select...")

Query q = em.createNativeQuery("Select \* FROM inctest3(:val);");

q.setParameter("val", 3);

int res = (Integer)q.getSingleResult();

* Procedury można też wywoływać przez Session:
  + Wywoływać **KONIECZNIE w @Transactional**, bo inaczej będzie Exception

SessionFactory sf = emf.unwrap(SessionFactory.class);

Session s = sf.openSession();

ProcedureCall q = s.createStoredProcedureCall("p73");

q.registerParameter(1, Integer.class, ParameterMode.IN);

q.setParameter(1, 7);

ProcedureOutputs po = q.getOutputs();

ResultSetOutput resultSetOutput = (ResultSetOutput) po.getCurrent();

/ //Zwraca List. Można rzutowanć List<Object[]> lub List<List<Integer>>

List<List<Integer>> results = (List<List<Integer>>) resultSetOutput.getResultList();

* Procedury można też wywoływać przez repo
  + Wywoływać **KONIECZNIE w @Transactional**, bo inaczej będzie Exception

@Procedure

public Integer p61(@Param("i") Integer i);

…

Integer res = t1d.p61(5)

* Procedury w postgresie mogą zwracać różne typy danych:
  + void
    - Hibernate nie lubi void, więc createStoredProcedureQuery(inctest2) powinna zamiast execute() robić ...getFirstResult(), co pozwala uniknąć EX: No Dialect mapping...
    - Zadziała też … createNativeQuery("Select 1 FROM inctest2(:val);")
  + pojedynczą wartość
    - createNativeQuery
    - createStoredProcedureQuery
    - createNamedStoredProcedureQuery
    - createStoredProcedureCall
  + zwracanie wiele wartości
    - z użyciem RETURNS refcursor - w parametrach nie jest obecny parametr OUT
      * createStoredProcedureQuery
    - z użyciem RETURNS TABLE(o1 integer, o2 integer) - w parametrach nie jest obecny parametr OUT
      * createStoredProcedureCall
    - pozostałe sposoby nie działają (np. createNamedStoredProcedureQuery)
* Używanie procedur w zapytaniach JPA jest raczej niemożliwe.

Wersjonowanie

Dodanie kolumny z adnotacją @Version typu integer. Ogólnie działanie wygląda tak, jakby sprawdzany był parent wraz z WCZYTANYMI dziećmi i porównywane były znalezione wartości z pól val.

* Insert wstawia wartość 0
* automatycznie podbija licznik na każdy Update
* Jeśli obiekt posiada dzieci z polem @Version, to update dziecka A spowoduje Exception, nawet jeśli wykonujemy update dziecka B lub parenta (dziecko A zostawiamy bez zmian), o ile to dziecko było wczytane (czyli dla Lazy dziecko może być null i wtedy błąd nie wystąpi)

Spring i transakcje

W Springu transakcje realizuje się przez @Transactional, które mozna dodawać nad klacę/inferface, albo nad metodą.

* ReadOnly = true (def. false)
  + JPA będzie mógł zastosować pewne optymalizacje (np powoduje pominięcie dirty check - czy obiekt wymaga aktualizacji)
  + Powinien być stosowany do S, ale NIE sprawdza, czy jest UPDATE lub INSERT, (niektóre DB odrzucają I czy U podczas transakcji RO)

|  |
| --- |
| Przykład repo z JPARepository i CustomRepo |
| //Niezbędne do automatycznego tworzenia repo z użyciem [Crud/JPA]Repository  @EnableJpaRepositories(basePackages = { "pl.biu" })  ...  **public** **interface** CompanyRepo **extends** JpaRepository<Company, Long> , CompanyCustomRepo  {  @Query(”…where d.code = ?1…")  **public** Long findIdByCode(String code, );  @Query(”…where d.id = :id…")  **public** Long findIdByCode(@Param("id") @NotNull Long idParam);  @Query("Select p.age, p.name …")  **public** List<Tuple> getTupleById();//res.getTupleById(0).get(0, Integer.**class**)  @Query("Select p From Person p")  **public** List<Person> getJpaList();//List<Person> lst = res.getJpaList();  @Query("Select p From Person p")  **public** Stream<Person> getJpaStream();//r.getJpaStream().iterator();  …  }  **public** **interface** CompanyCustomRepo {  **public** String getName();  …  }  **public** **class** CompanyCustomRepoImpl **implements** CompanyCustomRepo {  @Autowired EntityManager em;  // Pozwala pobrać aktualną sesję hibernate  **private** Session getCurrent() {  **return** em.unwrap(Session.**class**);  }  @Override  **public** Person getName () {  Query q = em.createQuery("Select p From Person WHERE p.id = :id");  q.setParameter("id", 10000000L);  **return** q.getResultList().getIdx(0); // Pobieranie listy wyników  **long** val = ((Number)q.getSingleResult()).longValue();//Pobiera jednego wiersza rzuca [NoResultException](eclipse-javadoc:%E2%98%82=Hibernate_02/C:%5C/Users%5C/mudia.000%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.hibernate.javax.persistence%5C/hibernate-jpa-2.1-api%5C/1.0.2.Final%5C/52afb5762c704a6b586e27742470c08f91877fc1%5C/hibernate-jpa-2.1-api-1.0.2.Final.jar%3Cjavax.persistence(Query.class%E2%98%83Query~getSingleResult%E2%98%82NoResultException) [NonUniqueResultException](eclipse-javadoc:%E2%98%82=Hibernate_02/C:%5C/Users%5C/mudia.000%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.hibernate.javax.persistence%5C/hibernate-jpa-2.1-api%5C/1.0.2.Final%5C/52afb5762c704a6b586e27742470c08f91877fc1%5C/hibernate-jpa-2.1-api-1.0.2.Final.jar%3Cjavax.persistence(Query.class%E2%98%83Query~getSingleResult%E2%98%82NonUniqueResultException)  }  // Odda listę obieków Person możliwości dokładnie takie jak @Query  // Stream UŻYWA scroll() I ScrollableResult Z JDBC i pozwala pobierać pojedyncze wiersze.  // Nie ma to wpywu ma samo zapytanie, więc nie można używać, do optymalizacji" jak ROW, ale  // pozwala na pobieranie dużych danych po kawałku  @Override  **public** Stream<String> getNameStream() {  Session ses = getCurrent();  Stream<String> res = ses.createQuery("Select p.name …", String.**class**).stream();  **return** res;  }  @Override  **public** List<Object[]> countByFormLocationId(Long formLocationId)  {  Query q = em.createNativeQuery("…");  q.setParameter("id", id);  List<Object[]> res = q.getResultList();    **return** res;  }  } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |