<https://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/chapters/query/native/Native.html>

**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
    - nie pozwoli na wykonanie SELECT przez T2, dopóki T1 nie zakończy się
  + Użyc 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

**Cache**

* Cache L1 jest powiązany z sesją i wraz z nią jest czyszczony.
* Cache L2 Cache wystaje poza sesję i jest współdzielonyz zinnymi sesjami
* 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>

|  |
| --- |
| Obsługa cache L1 |
| //Usuwanie danych z L1  s.evict(*entity*);  s.clear();  //Sprawdzanie istnienia w L1 – sprawdzanie po hash, więc inne entity o takim samym id (klon) odda false  session.contains(*entity*);  // Statystyki L1  SessionStatistics ss = s.getStatistics();  // Klucze w L1  Set<EntityKey> sk = ss.getEntityKeys();  // Iteracja po kluczach L1  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()) |

|  |
| --- |
| Obsługa cache L2 |
| //Pobieranie cache  Cache cache = em.getEntityManagerFactory().getCache();  Cache cache = sf.getCache()  //Usuwanie danych z L2  cache.evict(Person.**class**, *personPK*);  cache.evict(Person.**class**);  cache.evictAll();  //Sprawdzanie istnienia w L1 – sprawdzanie po hash, więc inne entity o takim samym id (klon) odda false  cache.containsEntity( Test1.**class**, t1.getId() ) |

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

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

…

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

**query.setCacheable(true);**

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

**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)- podstawowe cechy i budowa

* **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 argumentowy
* JPA odwołuje się bezpośrednio do pól (public) albo do getX i setX (GS)
* @E może extends ~~@E~~ i na odwrót, ale pola z ~~@E~~ nie są używane przez H
* 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)
    - @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 – listy, kolekcje, embedded  @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<>(); |
|  |
|  |
| Delete vs BulkDelete (hibernate03)  DB (poniżej) nie będzie wiedzieć, że obiekt usunięto i może oddać go z Cache. DB zadziałą poprawnie bo bazuje na em.Remove(...)  @Modifying  @Query("DELETE FROM Test1 WHERE id=?1")  **public** **void** bulkTest1Delete(Long id); |
| Merge po Detach  em.persist(t1); // zapisze obiekt do cache  em.flush(); // zrzuci cache do bazy  em.detach(t1); // odłaczy obiet od em  t1.setIntVal1(99); // odłączony obiekt nie wpłynie na cache  t3 = em.merge(t1); // podłączy obiekt do em (wykona SELECT) i odda obiekt z PC. Jeśli t1 był w PC to t1===t3, wpp t1 będzie tylko nośnikiem danych i nie będzie połaczony z PC, ale lepiej na tym nie polegać i DZIAŁAĆ NA t3!  em.flush(); // wykona UPDATE obieky w DB |
| Detach i ponowny Persist - ERR  em.persist(t1); //zapisze obiekt do cache  em.flush(); //zrzuci cache do bazy  em.detach(t1); //odłaczy obiet od em  t1.setIntVal1(99); //odłączony obiekt nie wpłynie na cache  em.persist(t1); //PK VIOLATION  em.persist(t1);  em.flush();  t1.setIntVal1(98);  Test1 t2 = em.find(Test1.**class**, t1.getId());  ---  em.detach(t2);  t2.setIntVal1(99);  em.persist(t2); - INSERT – Błąd, Duplicate PK  ---  t2.setIntVal1(99);  em.persist(t2); - OK |
| Obiekt Detach I nieDetach są różne  Test1 t2 = em.find(Test1.**class**, t1.getId());  Test1 t3 = em.find(Test1.**class**, t1.getId());// t3===t2  em.detach(t2); // t2 i t3 są detached!  ---  Test1 t2 = em.find(Test1.**class**, t1.getId());  em.detach(t2); // t2 jest detached!  Test1 t3 = em.find(Test1.**class**, t1.getId());// t3 !== t2 |
| Pobranie od H obektu wskazuje na ten sam obiekt  em.persist(t1);  t1.setIntVal1(91);  Test1 t2 = em.find(Test1.**class**, t1.getId()); // t2 = 91  t2.setIntVal1(92); // t2 = 92, t1 = 92 |
| Session save vs saveOrUpdate  Session s = em.unwrap(Session.**class**);  Test1 t2 = em.find(Test1.**class**, t1.getId());  em.detach(t2);  t2.setIntVal1(88);  //s.save(t2); //PK VIOLATION – jak Persistance  s.saveOrUpdate(t2); //Poprawny UPDATE – jak merge |
| Procedure  “inctest2” To nazwa metody w SQL  HQL  StoredProcedureQuery q = em.createStoredProcedureQuery("inctest2");  q.registerStoredProcedureParameter(0, Integer.**class**, ParameterMode.***IN***); //Parameter przez numer  q.setParameter(0, Integer.*valueOf*(2));  //q.execute(); - H nie lubi jak procedura oddaje dummy, więc modyfikacja procedury albo getFirstResult();  q.getFirstResult();  Native  Query q = em.createNativeQuery("Select 1 FROM inctest2(:val);"); //Zwracanie wartości dummy  Query q = em.createNativeQuery("Select \* FROM inctest3(:val);");//Zwracanie wyniku  q.setParameter("val", 3);  **int** res = (Integer)q.getSingleResult();  Wyniki przez Out – raczej nie ma opcji dla native  StoredProcedureQuery q = em.createStoredProcedureQuery("p62");  q.registerStoredProcedureParameter("i", Integer.**class**, ParameterMode.***IN***); //Parameter przez nazwę parametry  q.registerStoredProcedureParameter("o", Integer.**class**, ParameterMode.***OUT***);  q.setParameter("i", Integer.*valueOf*(4));  // wywołanie któregokolwiek z poniższych da "Unable to extract OUT/INOUT"  //boolean gotRes = q.execute();  //List<Object[]> postComments = q.getResultList();  **int** res = (Integer)q.getOutputParameterValue("o"); // res == 5  Named procedures – działa dokładnie jak powyżej  @NamedStoredProcedureQueries(value = {  @NamedStoredProcedureQuery(  name = "hql\_62", procedureName = "p62", parameters = {  @StoredProcedureParameter(name = "i", mode = ParameterMode.***IN***, type = Integer.**class**),  @StoredProcedureParameter(name = "o", mode = ParameterMode.***OUT***, type = Integer.**class**)  }),  …})  StoredProcedureQuery q = em.createNamedStoredProcedureQuery("hql\_62");  q.setParameter("i", 5);  Object r = q.getOutputParameterValue("o");// res == 6  Named procedures – zwraca: RETURN QUERY select t.val, i FROM tbl t;  @NamedStoredProcedureQuery(  name = "it73",  procedureName = "inctest73",  parameters = {  @StoredProcedureParameter(name = "i", mode = ParameterMode.***IN***, type = Integer.**class**)  }  )  StoredProcedureQuery q = em.createNamedStoredProcedureQuery("it73");  q.setParameter("i", 6);  List<Object> res = q.getResultList();//Zwraca liaste wierszy(lista object) |
| Projekcje – takie coś jak View z DB  **public** **interface** SubObj {  **public** Integer getV3(); //… as v3 -> getV3()  **public** Integer getint\_val(); //… .int\_val -> getint\_val()  }  ...  /\* "getint\_val1" DZIAŁA.  \* "getInt\_val1" NIE DZIAŁA.  \*/  @Query(value="SELECT t3.int\_val1 as v3, t2.int\_val FROM Test3 t3 LEFT JOIN Test2 t2 ON t3.t2\_id = t2.id", nativeQuery = **true**)  **public** List<SubObj> getSubObj(); |
| Sort - Dodaje "Order By ..."  @Autowired  Test1Dao t1d;  List<Integer> res = t1d.getSorted(**new** Sort(Sort.Direction.***DESC***, "id"));  List<Integer> res = t1d.getSorted(**new** Sort(Sort.Direction.***ASC***, "t2.id"));//Sortowanie po polu podobiektu |
| Scrollable  SessionFactory sf = emf.unwrap(SessionFactory.**class**);  StatelessSession s2= sf.openStatelessSession();//Session s2 = sf.openSession();  ScrollableResults r = s2.createQuery("SELECT t1 FROM Test1 t1").scroll(ScrollMode.***FORWARD\_ONLY***);  **while** (r.next()) {  Test1 t1 = (Test1) r.get()[0];  unwrap} |
| Translock  @Entity  **public** **class** Test71 {  ...  @Version  **public** **int** ver;  Test70 t7 = t7d.findById(Long.*valueOf*(1)).get();  /\*Tu następuje zmiana wartości poza transakcją\*/  t7.setVal(12);  t7d.save(t7);  /\*Tu rzuca wyjątkiem ObjectOptymisticLockingException\*/ |
|  |

* [O/O][M/M] ->
* @Embedded (niby) nie jest konieczne (ale dobrze wygląda), ale w rzeczywistości nie doszło do kompilacji
* Ta sama klasa nie może być użyta do @E i @OO etc
* @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 ->
* @E może dziedziczyć po nie @E, ale zawartość ta jest pomijana przez JPA -> @Inheritance
* @Inheritance - opisuje konstruowanie tabel w przypadku dziedziczenia ->
* @ID - może być basic lub obiektem klasy –>
* 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>

**Converter**

|  |
| --- |
| Converter |
| @Converter(autoApply = **true**)  **public** **class** TestConverter **implements** AttributeConverter<String, Short>/\*String w java, short w DB\*/  {  **public** Short convertToDatabaseColumn(String enityVal)  {...}  **public** T convertToEntityAttribute(Short dbVal)  {...}  }  Przykład z Enum  @Converter(autoApply = **true**  **public** **class** TestConverter **implements** AttributeConverter<FormStateStatus, Short>  {  **public** Short convertToDatabaseColumn(FormStateStatus enityVal)  {...}  **public** FormStateStatus convertToEntityAttribute(Short dbVal)  {...}  }  Generycznty przykład z Enum  @Converter(autoApply = **true**  **public** abstract **class** EnumWithShortCodeConverter<T **extends** Enum<?> & HasShortCodeAndLabel> **implements** AttributeConverter<T, Short>  {  **public** Short convertToDatabaseColumn(T enityVal)  {...}  **public** T convertToEntityAttribute(Short dbVal)  {...}  }  **public** **class** FormStateStatusConverter **extends** EnumWithShortCodeConverter<FormStateStatus>  @Column  @Convert(converter = FormStateStatusConverter.**class**)  **public** String/\*FormStateStatus lub inny T **extends** Enum …\*/ status; |

**Kolekcje / Relacje**

* Mapowanie @OneToMany, @ManyToMany i @ElementCollection jest wygodne z punktu widzenia budowy obiektów, ale przez używanie FK jest kosztonwe. Dla małych kolekcji koszt może być mały, ale dla dużych zauważalny.
* Problemem bywa nie tylko liczba obiektów w kolekcji, ale także liczba samych kolekcji, bo jedna kolekcja to porencjalnie osobne query.
* Lazy czesto nie jest wyjściem, bo i tak musi zostać wczytana lista, aby otrzymać np rozmiar i przypadkiem (przy czytaniu rozmiaru) możemy przypadkiem „dotknąć” kolejcji.
* Ogólnie, zawsze szybszy bedzie dostęp przez wykonanie dedykowanego query w przypadku kolekcji.
* Junction Table: tabela łącząca klucze 2 tabel. @JoinTable(name,joinColumns, inverseJoinColumns)
  + joinColumns = @JoinColumn(name = „postId”) – id 1 tabeli
  + inverseJoinColumns = @JoinColumn(name = „postId”) – id 2 tabeli
* Najlepsze mapowiania @ManyToMany i @OneToOne są te kontrolowane przez child
* 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, co powoduje znaczne zwiększenie liczby zapytań.
  + 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
* 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(parent)”) - powoduje, że dla OO obie strony mają ten sam klucz, ale dla @MO nie wiem co o tym sądzić
* TRZEBA pamiętać o cascade lub wcześniej zapisać obiekt (w tej samej transakcji?) wpp. Exception. Cascade powinno być po stronie zapisywanego obiektu – zazwyczaj to parent.
* <http://in.relation.to/2016/09/28/performance-tuning-and-best-practices/>

@ManyToOne

* FK będzie po stronie dziecka (MO) dla **Uni** i **Bi** - podejście podobne jak w SITU (unSprawa)
* Szybkie i sprawne
* @ManyToOne(fetch = FetchType.***LAZY***) - Pozwala na Lazy
  + @MapsId
  + @JoinColumn(name = „idRodzica”)
* @JoinColumn(...) – nazwa kolumny z id rodzica
* @MapsId – po stronie dziecka, pobiera id od rodzica i przepisuje na własne id

@ MapsId („xxx”) wskazuje pole w kluczu, do którego należy wpisać klucz rodzica (bezsensu dla klucza złożonego z jednego pola). **Typ id rodzica i pola w id dziecka musi być zgodny.**

|  |
| --- |
| @ManyToOne |
| **// Klucz pozwala na tylko jedną cechę danego typu dla każdego parenta**  **public** **class** CechaId **implements** Serializable {  **public** Long idParent;  **public** String idCecha;}  …  **public** **class** Cecha {  @EmbeddedId CechaId cechaId;  ...  @ManyToOne(fetch = FetchType.***LAZY***)  @MapsId("idParent") // Pole idParent w kluczu będzie pobierane z @Id w Szkoda  **public** Szkoda szkoda; |

@OneToMany

* **Bi** będzie trzymać FK po strobie child – polecane. Musi posiadać **MappedBy**, czyli wskazanie. Które pole w dziecu wskazuje na rodzica
* **Uni** utworzy specjalną tabelę łączącą (Junction) klucze (chyba że jest @JoinColumn ) – niepolecane. Dodanie do kolekcji zrobi 2 inserty (samą pozycję kolekcji i tabeli JT).
* Najlepszą taktyką dla @OneToMany jest @ManyToOne Uni z @MapsId
* Po stronie parent musi mieć cascade lub wcześniej zapisać obiekt (w tej samej transakcji?) wpp. Exception
* @OrderColumn – (ma zmniajszać liczbę zapytań przy operacjach na liście) SKOMPLIKOWANE A I TAK WYCHODZI KICHA! dla Uni. Jeśli kolekcja jest indexowalna (List) dodaje kolumnę entry przechowującą kolejność (0,1,2) czyli index z Listy. Wtedy
  + Parent.list.remove(2); spowoduje
  + DELETE FROM … WHERE parentId=… AND entry = 2;
  + UPDATE zmniejszające entry dla elem. Powyżej usuwanego

DELETE FROM Child WHERE ...

* @JoinColumn(name = „post\_id”) – powoduje brak JT, ale dodaje kolumnę do dziecka (z id rodzica), tak jakby było BI – rozwiązanie podobne do SITU. NIESTETY! Pomimo budowy jak w BI, INSERT dziecka wpierw wypełnia oryginalne kolumny dziecka, a potem robi UPDATE wypełniając kolumnę z id rodzica – KICHA! Dziecko nic nie wie o tej kolumnie, więc choć tabele mają odpowiednią strukturę, to JPA dzieka nie wie o tej kolumnie.

DELETE w tym przypadku wpierw ustala idRodzic = null a potem dopiero robi delete.

|  |
| --- |
| @OneToMany |
| // Najprostsze Uni korzysta z tabeli pośredniczącej – 2 insery  @OneToMany(cascade = CascadeType.***ALL***, orphanRemoval = true)  // Uni korzystające z dodatkowej kolumny w child – 1 insert  @OneToMany(cascade = CascadeType.***ALL***, orphanRemoval = true)  @JoinColumn(name = ***" parent\_id\_fk"***) // Niekonieczne – nadaje nazwę FK  // Bi używające - wskazane  @OneToMany(cascade = CascadeType.***ALL***, mappedBy = “***parent”***, orphanRemoval = true)  …  @ManyToOne(fetch = FetchType.***LAZY***)  @JoinColumn(name = ***"parent\_id\_fk"***) // Niekonieczne – nadaje nazwę FK  **public** Parent parent;  // Można użyć @ManyToOne Uni - wskazane |
|  |

@ElementCollection

* takie ManyToOne dla typów prostych
* INSERT szybki, DELETE wolny, bo wiele DELETE całej tabeli i ponowy INSERT
* @OrderColumn – działa jak powyżej
* Elementami mogą być typy proste lub Obiekty
* Elementem może być obiekt @E, ALE UWAŻAĆ !!! To wymaga wpierw zapisu elementów kolekcji a potem samego person1

|  |
| --- |
| @ElementCollection |
| @Entity  **public** **class** Person1 {  @Id  **public** Long id;  …  //Feature1: @Embeddable –  // osobna tabela “person1\_features”: pola z Feature1 + 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 String + pole "person1\_id"  @ElementCollection  **public** List<String> strings = **new** ArrayList<>();  // Dokładnie j.w.  @ElementCollection(targetClass=String.**class**)  **public** List annoStrings = **new** ArrayList();  //Jak lista – “person1\_features” + pole "person1\_id" + pole “features\_key” –klucz mapy  @ElementCollection(fetch=FetchType.***EAGER***)  **public** Map<Integer, Feature1> features = **new** HashMap<>();  //Feature1: @Entity – DZIAŁA, ALE NIE UŻYWAC!!!  @ElementCollection(fetch=FetchType.***EAGER***)  **public** Map<Integer, Feature2> features2 = **new** HashMap<>(); |

@OneToOne

* **Najcześciej jest wykorzystywany UNI w dziecku z MapsId i Lazy.** Pozwala wczytać Rodzica i Dziecko osobno. Pozwala doczytać Rodzica w dziecku a rodzic i dziecko mają wspólną warość PK, więc mając jedno możemy doczytać drugie. Problemem jest nienaturalność budowy obiektów – rodzic nie ma pola z dzieckiem, choć można zasymoulować odpowiednim prywatnym polem i getterami/setterami.
* **Uni – FK zawsze po stronie rodzica**
* Jest bardzo podobne w działaniu do @ManyToOne. Właściwie można stosować zamiennie ALE ManyToOne wymaga kolejcji jako obiektu, co wygląda źle w kodzie.
* Jest tworzony za pomocę Unique FK
* Może być też tworzony z użyciem @JoinColumn, ale to wygląda na zwykłe nadawanie nazwy, a nie przeniesienie
  + @OneToOne(cascade = CascadeType.***ALL***)

**public** ChildOO1 child;

* + - Pojedynczy select Left Outer Join
  + @OneToOne(cascade = CascadeType.***ALL***)

MapsId

ChildOO1 child;

* + - Select child + select parent – niby współdzielenie jest lepsze ale wymaga 2 Select ???
* **BI**
  + Ogólnie jest problem z Lazy (zawsze jest doczytywane). Jedynym sposobem obejścia jest użcie Bytecode, ale to jest skomplikowane, więc przeferowany jest Uni.
  + Strona z MappedBy jest inverse, bez MB strony będa traktowane jak podwójny unidirectionall
  + @OneToOne(cascade = CascadeType.***ALL***)

ChildOO1 child;

…

@OneToOne

@MapsId

p**ublic** ParentOO1 parent;

* + - Select \* 2
    - **Dodając mappedBy w parent (= "parent") uzyskamy pojedynczy select Left Outer Join**
    - fetch = FetchType.Lazy – Select \* 2
  + @OneToOne(cascade = CascadeType.***ALL***)

ChildOO1 child;

…

OneToOne

@MapsId

p**ublic** ParentOO1 parent;

* + Select child + select parent
* Po stronie inverse/parent musi mieć cascade lub wcześniej zapisać obiekt (w tej samej transakcji?) wpp. Exception
* <https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/>

|  |
| --- |
| @OneToOne |
| **// Unidirectional - najlepsze**  @OneToOne  @MapsId  **public** Parent parent  **// Bidirectional - najpopularniejsze**  @OneToOne(mappedBy = ”parent”, cascade = CascadeType.***ALL***)  **public** Child child  ...  @OneToOne (fetch = FetchType.***LAZY***)  @MapsId  **public** Parent parent; |

@ManyToMany

* Domyślnie jest LAZY
* Ogólnie unikać List i lepiej w ogóle unikać @ManyToMany, bo można się zamotać.
* **UNI**
* Połączeni jest realizowane przez JT. To powoduje problem dla List (nie dla Set) (podobny do @OneToMany) przy usuwanie, bo wpierw czyszczona jest JT z wpisów dla wybranego klucza a potem potem na nowo wypełniana
* Caskadowanie – Z racjinatury ManyToMany lepiej nie używać REMOVE (i ALL), a wskazać wyłącznie PERSIST i MERGE
* **BI**
* @ManyToMany jest realizowane jako 2 **Uni** @OneToMany, co oznacza AŻ 2 JT!!
* Dla dziecka @ManyToMany(mappedBy=”tags”) spowoduje tylko 1 JT!
  + @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.

|  |
| --- |
| @ManyToMany |
| **public** **class** Tag {  @Id  private Long id1;  @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})  @JoinTable(name = "post\_tag", // Opisuje kolumny, po których będzie łączenie w JT  joinColumns = @JoinColumn(name = "post\_id1"),  inverseJoinColumns = @JoinColumn(name = "tag\_id2")  )  private List<Tag> tags = new ArrayList<>();  ...  **public** **class** Tag {  @Id  private Long id2;  @ManyToMany(mappedBy = "tags")  private List<Post> posts = new ArrayList<>(); |

**ID**

* @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

|  |
| --- |
| @Id (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; |

* Klucze złożone: Composite @Id i @EmbeddedId potrzebują klasy Serializable

|  |
| --- |
| Composite @Id vs @EmbeddedId |
| 1 robi osobną tabelę i kolumnę wskazującą na wpis w tamtej, a 2 robi normalny klucz 2 kolumnowy  @Embeddable  **public** **class** BookId **implements** Serializable {  **private** String author;  **private** String name;  }  @Entity  @IdClass(BookId.**class**)  **public** **class** BookComposite {  @Id String author; // Nazwa musi odpowiadać nazwie pola w BookId  @Id String name; // Nazwa musi odpowiadać nazwie pola w BookId  }  @Entity  **public** **class** BookEmbeddedId {  @EmbeddedId **private** BookId id;  } |

* **Generatory**
  + Typy
    - IDENTITY: (auto\_increment), związany tylko z 1 tabelą, H wykonuje tylko INSERT i pobiera wynik, brak batch
    - SEQUENCE: H wykonuje SELECT a potem INSERT. Używa batch (ale wciąż wolniejszy od I)
    - TABLE: sztuczne sequence gdy brak natywnego. Używa batch, ale wymaga 3 SQL, robi lock. Mniejsza optymalizacja dla wysokiej współbieżości. Więcej wątków, większa różnica.
  + @GeneratedValue – generator domyślny, osobny dla każdej tabeli
  + @GeneratedValue + @SequenceGenerator – Wskazujemy na konkretny generator, pozwala współdzielić generator pomiędzy encjami.
    - Przy współdzieleniu @GeneratedValue + @SequenceGenerator musża być przy każdej @E
    - Allocation size jest opcjonalne (domyślnie 1) – pula per encja. Przy współdzieleniu musi być wszędzie takie samo
    - @E zachowuje swoją pulę pomiędzy flush
    - Gdy wybrane, to będzie powodować powstawnie przerw (każda).
    - Pobieranie większej liczby wartości przyspiesza działanie DB
  + Ogólnie, nie znalazłem sposobu na wspólny generator, który pozwala na pobieranie pool, bez przerw. Albo pule z przerwami, albo pojedynczo, albo osobny mechanizm i ręczne przypisywaie Id.

|  |
| --- |
| Generatory |
| Jak używać sp\_sequence\_get\_range w H? Osobna transakcja?  // hh – nazwa łącząca adotacje  // h – nazwa sekwencji do utworzenia/użycia w DB  // allocationSize jest opcjoanalne  @GeneratedValue(generator = "hh", strategy=GenerationType.***SEQUENCE***)  @SequenceGenerator(allocationSize = 50, name = "hh", sequenceName = "h")  Ręczne definiowanie generatora  @GenericGenerator( name = "hhh", strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator"  , parameters = {  @Parameter(name = SequenceStyleGenerator.***CONFIG\_PREFER\_SEQUENCE\_PER\_ENTITY***, value = "false"),  @Parameter(name = "prefer\_sequence\_per\_entity", value = "false"),  // @Parameter(name = "sequence\_per\_entity\_suffix", value = "\_seq"),  // @Parameter(name = "initial\_value", value = "5000000"),  @Parameter(name = SequenceStyleGenerator.***INCREMENT\_PARAM***, value = "30"),  @Parameter(name = SequenceStyleGenerator.***SEQUENCE\_PARAM***, value = "hh"),  @Parameter(name = "allocation\_size", value = "30"),  }) |

**Dziedziczenie**

* @Inheritance - opisuje konstruowanie tabel w przypadku dziedziczenia:
  + - **Polimorfoczny query** (gdy chcemy wszystkie dziedziczące z bazowej) odda listę, ale każdy obiekt będzie miał własną sobklasę ☺
    - Ustalane dla klasy root - **NIE MOŻNA** zmianiać w łańcuhu dziedziczenia!
    - W łańcuch dziedziczenia **MOŻNA** wplatać @MappedSuperclass.
  + SINGLE\_TABLE - zbije wszystkie klasy w jedną tabelę.
    - Domyślne
    - **Zalecane gdy podklas jest mało i są nieduże róznice w polach**
    - **+ Szybki zapis (insert per klasa) i odczyt (Selecty bez JOIN)**
    - **+ Dobra obsługa PQ (nie wymaga JOIN)**
    - **– Pola z klas dziedziczących MUSZĄ być nullable.** Można to obejść używając walidacji Beanowej (@NotNull) lub metod callback z JPA (@PreUpdate) lub @EntityListeners
    - jest kolumna discriminatorColumn javax.persistence.DiscriminatorType:STRING(domyślnie nazwa klasy), CHAR, INTEGER opisująca klasę końcową
  + TABLE\_PER\_CLASS - każda końcowa klasa (ścieżka aż do root) we własnej tabeli (nie wszystkie implementacje JPA to obsługują). Odziedziczone pola są trzymane w wielu tabelach
    - + Szybki write
    - –/+ Woly read dla PQ, szybki dla klas końcowych
    - **– Kiepska obsługa PQ (wymagają wielu UNION)**.
  + JOINED - poszczególne klasy są trzymane w osobnych tabelach
    - **Zalecane gdy dużo podklas ale żadko wymagane PQ**
    - **+ Wszystkie pola mogą być NotNull**
    - **– Zwiększa się index footprint, bo jest ich więcej**
    - **– Kiepska obsługa PQ (wymagają wielu JOIN)**. Można to obejść za pomocą paginating, ale tylkow PQL ale już nie w @OM czy @MM. @OM i @MM nie zwiększą złożoności jeśli oddają tylko po jednym obiekcie.
    - **– Wymaga joinów do pobrania pojedynczej klasy (zwiększa czas kalkulacji planu)**
    - Pomijane są klasy bez @E. Reszta (nawet bez nowych pól) są uwzględniane
    - dobrze odwzorowuje polimorfizm, ale wymaga wielu JOIN aby wyciągnąć cały obiekt
  + @MappedSuperclass – nie jest opcją, ale podobne też dziedziczenie
    - nie ma tabeli, nie może być query, nie może być częścią relacji
    - Dobre do oddzialenia struktury obiektów i BD
    - - Brak polimorfizmu
    - + Szybki odczyt i zapis

|  |
| --- |
| Inheritance - Dziedziczenie (hibernate02.inheritance) |
| @Entity  //@Inheritance – TYLKO NAD ROOT, nie domyślnie (nie potrzeba @) SINGLE\_TABLE  @Inheritance// default - SINGLE\_TABLE  //@DiscriminatorColumn(discriminatorType = DiscriminatorType.STRING, name = "TYPE") – opcjonalna kolumna określająca z którego obiektu pochodzą dane – używane dla JOINED i SINGLE\_TABLE. Spoza hibernate będziemy wiedzieć jakiego typu są dane. Tylko dla Root  **public** **class** Person\_gen {  @Id  @GeneratedValue  **private** Integer id;  **private** String stGen;  }  ...  //@DiscriminatorValue("BIRD") – opcjonalne – nadawane są wartości domyślne – zależy od providera  @Entity  **public** **class** Person **extends** Person\_gen{  **private** String st;  }  ...  @Entity  **public** **class** PersonInWork **extends** Person {  **public** String stInWork;  }  JOINED - Każdy poziom ma id oraz własne pola - obiekt jest rozpostarty na wielu tabelach  SINGLE\_TABLE - Jest jedna tabela root "Person\_gen", która posiada sumę wszystkich kolumn (z dziedziczenia)  TABLE\_PER\_CLASS - Każdy poziom ma osobną tabelę - zawiera sumę kolumn z dzidziczonych tabel  Jeśli mamy ciąg En(a)->En(b)->(c)->MapSup(d)->En(e)->Em(f), to będą kolumny a,b,e  Jeśli mamy ciąg En(a)->En(b)->(c)->~~Em(d)~~->En(e)->Em(f), „Subclass has to be binded after it's mother class: (d)”  @MappedSuperClass// - „Wirtualna” encja na potrzeby dziedziczenia javy - W bazie będą tylko Person i PersonInWork posiadające sumę kolumn  **public** **class** Person\_gen {  @Id  @GeneratedValue  **private** Integer id;  **private** String stGen;  }  ...  @Entity  **public** **class** Person **extends** Person\_gen{  **private** String st;  }  ...  @Entity  **public** **class** PersonInWork **extends** Person {  **public** String stInWork;  } |

**Session**

|  |
| --- |
| Session |
| //Pobieranie Session i SessionFactory w Springu  Session s = em.unwrap(Session.**class**); //@Autowired EntityManager em  SessionFactory sf = emf.unwrap(SessionFactory.**class**); //@Autowired EntityManagerFactory emf |

* Operacje dla Session
  + 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
  + save (tylko H) - jak persist, ale jeśli obj został detached, to będzie INSERT zamiast UPDATE
    - Save zapisze do bazy od razu (także poza transakcją) a Persuist tylko w czasie commit lub flusf (pod transakcją)
    - Save odda obiekt z PC a Persist null
  + saveOrUpdate – jak save, ale wpierw stara się zrobić UPDATE, więc nie będzie PK VIOLATION
  + 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);

Jeśli person było w PC (wewnątrz transakcji?) to em.merge(person) podownie połączy do z PC

* + update - jak merge, ale nie zwraca nowego, tylko uaktualnia person. Jeśli person nie ma w DB, to rzuci PersistenceException
* Persist i Merge są z API JPA a ich odpowidniki Save i Update są „oryginalnymi” metodami hibernate. Choć mają robić to samo, to różnią się implementacją i przez to mają inne efekty.
* 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)

**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.
  + -> **HQL i native w Springu**
* 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.

Jeśli person było w PC (wewnątrz transakcji?) to merge podownie połączy do z PC

* + **detach** - odłacza obiekt od PC. Wykonany na obiekcie w PC ale nie w DB da EXCEPTION
  + **getReference** – Oddaje obiekt (class ….ParentMO1), który ma zainicjalizowany TYLKO PK gdy obiekt jest w db, lub obiekt proxy (class ... .ParentMO1\_$$\_jvstfa1\_1b) gdy go brak. Niby nie robimy select, ale podczas persist czy selete SELECT i taqk jest wykonywany – zero zysku. NATOMIAST przydatne, gdy
    - chcemy użyć tego obiektu jako warunku where
    - chcemy „dodać” referencję tego obieku do innego.

Wtedy nie robi SELECT i mam zysk!

https://www.baeldung.com/jpa-entity-manager-get-reference

|  |
| --- |
| EntityManager |
| em.find(xxx.**class**, pk);  em.persist(*entity*);  em.remove(*entity*);  em.refresh(*entity*);  em.flush();  Person mergedPerson = (Person) em.merge(*entity*);  em.detach(*entity*)  Game game1 = em.getReference(Game.**class**, 1L);  Player player1 = entityManager.find(Player.**class**, 1L);  player1.setGame(game1);  em.persist(player1); |

* @NamedQuery

@NamedQueries( {@NamedQuery(...), @NamedQuery(...)} )

|  |
| --- |
| @NamedQuery |
| @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 |

* NativeQuery
  + 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

|  |
| --- |
| Native query |
| Castowanie NativeQuery na typy proste  Query q = em.createNativeQuery("SELECT 1 FROM test2"/\*, Integer\*/);  Integer res = (Integer) q.getSingleResult();  **int** t = 0;  Castowanie NativeQuery na obiekty  /\* W przypadku, gdy zwracamy obiekt inny niż typy prostego, to należy podać klasę,  \* bo istnienie tego parametru rzutuje na budowę obiektu res i może uniemożliwiać  \* pózniejsze castowanie.  \*/  Query q = em.createNativeQuery("SELECT \* FROM test3 a", Test3.**class**);  /\* EAGER - Skutkiem ubocznym parametru "Test3.class" i tego że OO jet **EAGER** (default) jest to, że  \* doczytane zostaną podobiekty Test2. Dla każdego Test3 osobny SQL pobierający Test2.  \* Doczytanie będzie miało miejsce podczas wykonywania getResultList.  \*/  List<Object> o = q.getResultList();  Test3 res = (Test3) o.get(0);  // LazyInitializationException !!! – wczytywanie poza transakcją  String s = res.getT2().get(0).getSubStVal1(); |

**Mapowanie customowe**

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

|  |
| --- |
| Customowe mapowanie |
| @Entity  @SqlResultSetMappings( {  /\*  \* Mapowanie do pojedynczego obiektu @Entity  \*/  @SqlResultSetMapping(  name = "Test5Mapping\_1",  entities = {  @EntityResult(  entityClass = Test5.**class**,  fields = {  @FieldResult(name = "id", column = "id"),  @FieldResult(name = "intVal1", column = "val2")  })  }  )  ,  // Mapowanie z użyciem konstruktora  @SqlResultSetMapping(  name = "Test5Mapping\_3",  classes={  @ConstructorResult(  targetClass=Test5\_1.**class**,  columns={  //Kolejność pól MUSI BYĆ ZGODNA z kolejością arg. konstruktora  @ColumnResult(name="id", type=Long.**class**),  @ColumnResult(name="val2", type=Integer.**class**),  @ColumnResult(name="s1", type=Integer.**class**),  @ColumnResult(name="s2", type=String.**class**)  }  )  }  )  })  ...  **public** Test5(**long** id, **int** intVal1, **int** s1, String s2) {...}  ---  Query q = em.createNativeQuery("SELECT a.id, a.int\_val1 as val2, b.test5\_id as id2, b.sub\_int\_val1 as s1, b.sub\_st\_val1 as s2 FROM test5 a LEFT JOIN test5\_t2 b ON a.id = test5\_id", "Test5Mapping\_1");  List<Test5> o = q.getResultList();  Test5 res = o.get(0); |

**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.
* HQL można wywoływać w repo lub innych miejscach programu (Query q = em.createQuery…)
* 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 kodu musi być pod transakcja
* @Async pozwala na asynchroniczne wywołanie zapytania, które, jeszcze nie musiało się skończyć. Wynikiem jest „obietnica” obiektu.
  + Future<Obj>
  + CompletableFuture<Obj>
  + ListenableFuture<Obj>

|  |
| --- |
| Repository |
| //Niezbędne do automatycznego tworzenia repo z użyciem [Crud/JPA]Repository  @EnableJpaRepositories(basePackages = { "pl.biu" })  ...  **public** **interface** CompanyRepo **extends** JpaRepository<Company, Long /\*id extends Serializable\*/> , CompanyCustomRepo  {  @Query(”…where d.code = ?1…")  **public** Long findIdByCode(String code, );  // Parmetr nadajemy nazwę  @Query(”…where d.id = :id…")  **public** Long findIdByCode(@Param("id") @NotNull Long idParam);  // Przekazujemy cały obiekt jaka parametr i odwołumeny się do jego pół: #{#ba.id}  @Query("select b.id from BankAccount b where :#{#ba.id} is null or b.id <> :#{#ba.id}"  **public** List<Long> findConflictingPoolIds(@Param("ba") BankAccount ex);  // Zwracamy wyniki jako Tuple  @Query("Select p.age, p.name …")  **public** List<Tuple> getTupleById();//res.getTupleById(0).get(0, Integer.**class**)  // Zwracamy listę obiektów  @Query("Select p From Person p")  **public** List<Person> getJpaList();//List<Person> lst = res.getJpaList();  // Zwracamy stram obiektów  @Query("Select p From Person p")  **public** Stream<Person> getJpaStream();//r.getJpaStream().iterator();  // Zwracamy stram obiektów  @Async  @Query("Select p From Person p")  **public** CompletableFuture<Person> findById();//  …  }  // Custop repository interface  **public** **interface** CompanyCustomRepo {  **public** String getName();  …  }  // Custop repository implementation  **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;  }  } |

* Dziedziczenie/rozszerzanie interface

@NoRepositoryBean wyklucza interface z obsługiwania go przez Springa. Ten interface będzie mógł jednak być rozszerzany

* 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** **class** CustomDaoImpl **implements** CustomDao

**public** **interface** TestDao **extends** CrudRepository<T, Long>, CustomDao

**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
  + 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 i wystąpi 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.

|  |
| --- |
| Pagination |
| @Query("Select p From Person p")  **public** Page<Person> getPaged(Pagable pagable);//List<Person> lst = res.getJpaList();  ...  Page<Person> tmpRes = t1d.getPaged(pageable);  List<Person> lst = tmpRes.getContent();  Iterator<Person> person = tmpRes.iterator();  @Query(value = "Select p From Person p JOIN...",  countQuery = "Select count(p) From Person p JOIN...")  **public** Page<Person> getPaged(Pagable pagable); |

**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
* Płaski entity graph
* Entity graph z subgraphem
  + Subgraph opisuje jak zachowyją się list w dzieciach
  + Subgraph może korzystać 2 kolejnego subgraphu (opisującego wczytywanie wnuków). Poniżej SG „inLista” korzysta z SG „inLista2”

|  |
| --- |
| **Entity graph** |
| **// Składnia**  @NamedEntityGraph(  name = "graph.TestSix.lista",  attributeNodes = {@NamedAttributeNode("lazyList"), ...})  @NamedEntityGraph(  name = "graph.TestSix.lista",  attributeNodes = @NamedAttributeNode(value = "lazyList", subgraph = "inLista"),  subgraphs = {  @NamedSubgraph(name = "inLista", attributeNodes = {  @NamedAttributeNode(value = "list", subgraph = "inLista2")}),  @NamedSubgraph(name = "inLista2", attributeNodes = {  @NamedAttributeNode("list2")})  }  )  @NamedEntityGraphs ({ [@NamedEntityGraph]+ })  **// Wywoływanie**  @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")  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); |

* 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**

* hibernate04
* <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();

* Pobieranie sesji zwykłąej i Stateless:
  + StatelessSession – nie implementune C1 i nie korzysta z C2, automatycznego dirtychecking i ignoruje kolekcje. Po prostu nie ma większości mechanizmów H.
  + SessionFactory sf = emf.unwrap(SessionFactory.**class**);

Session s2= sf.openSession();//

* + StatelessSession s2= sf.openStatelessSession();//
* 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.
* Kod:

|  |
| --- |
| Procedure  “inctest2” To nazwa metody w SQL  HQL  StoredProcedureQuery q = em.createStoredProcedureQuery("inctest2");  q.registerStoredProcedureParameter(0, Integer.**class**, ParameterMode.***IN***); //Parameter przez numer  q.setParameter(0, Integer.*valueOf*(2));  //q.execute(); - H nie lubi jak procedura oddaje dummy, więc modyfikacja procedury albo getFirstResult();  q.getFirstResult();  Native  Query q = em.createNativeQuery("Select 1 FROM inctest2(:val);"); //Zwracanie wartości dummy  Query q = em.createNativeQuery("Select \* FROM inctest3(:val);");//Zwracanie wyniku  q.setParameter("val", 3);  **int** res = (Integer)q.getSingleResult();  Wyniki przez Out – raczej nie ma opcji dla native  StoredProcedureQuery q = em.createStoredProcedureQuery("p62");  q.registerStoredProcedureParameter("i", Integer.**class**, ParameterMode.***IN***); //Parameter przez nazwę parametry  q.registerStoredProcedureParameter("o", Integer.**class**, ParameterMode.***OUT***);  q.setParameter("i", Integer.*valueOf*(4));  // wywołanie któregokolwiek z poniższych da "Unable to extract OUT/INOUT"  //boolean gotRes = q.execute();  //List<Object[]> postComments = q.getResultList();  **int** res = (Integer)q.getOutputParameterValue("o"); // res == 5  Named procedures – działa dokładnie jak powyżej  @NamedStoredProcedureQueries(value = {  @NamedStoredProcedureQuery(  name = "hql\_62", procedureName = "p62", parameters = {  @StoredProcedureParameter(name = "i", mode = ParameterMode.***IN***, type = Integer.**class**),  @StoredProcedureParameter(name = "o", mode = ParameterMode.***OUT***, type = Integer.**class**)  }),  …})  StoredProcedureQuery q = em.createNamedStoredProcedureQuery("hql\_62");  q.setParameter("i", 5);  Object r = q.getOutputParameterValue("o");// res == 6  Named procedures – zwraca: RETURN QUERY select t.val, i FROM tbl t;  @NamedStoredProcedureQuery(  name = "it73",  procedureName = "inctest73",  parameters = {  @StoredProcedureParameter(name = "i", mode = ParameterMode.***IN***, type = Integer.**class**)  }  )  StoredProcedureQuery q = em.createNamedStoredProcedureQuery("it73");  q.setParameter("i", 6);  List<Object> res = q.getResultList();//Zwraca liaste wierszy(lista object) |

**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)
* Kod:

|  |
| --- |
| Translock  @Entity  **public** **class** Test71 {  ...  @Version  **public** **int** ver;  Test70 t7 = t7d.findById(Long.*valueOf*(1)).get();  /\*Tu następuje zmiana wartości poza transakcją\*/  t7.setVal(12);  t7d.save(t7);  /\*Tu rzuca wyjątkiem ObjectOptymisticLockingException\*/ |

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)

|  |
| --- |
| Batch insert/update |
| Batch otwiera coś w rodzaju kanału przepychając dane wg jednego wzorca (ten sam SQL ale inne dane).  # proponowana wielkosc 5-30. 0 -> domyslny, wylacza batch  spring.jpa.properties.hibernate.jdbc.batch\_size=30  spring.jpa.properties.hibernate.order\_inserts=true  spring.jpa.properties.hibernate.order\_updates=true  spring.jpa.properties.hibernate.jdbc.batch\_versioned\_data=true   * Batch zadziała dla INSERT typu SEQUENCE, ale nie IDENTITY. I wymagają pobrania wygenerowanego klucza z bazy, a tego H nie potrafi robić w batch. UPDATE działa niezależnie od I/S * Batch wykonywany jest w momencie flush (zbiorczo), wić można dodawać polecenie zapisu w róznych momentach transakcji, a i tak zostanie to zebrane w batch * P6spy wymaga pliku spy.properties (może być pusty) z linią: excludecategories=info,debug,result,resultset~~,batch~~   Batch przyspiesza zmniejszając liczbę wykoanywanych zapitać  B0 – 10k I, 10K S  357565753 nanoseconds spent preparing 30000 JDBC statements;  25297946912 nanoseconds spent executing 30000 JDBC statements;  0 nanoseconds spent executing 0 JDBC batches;  B30 – 10k I, 10K S  327790947 nanoseconds spent preparing 20001 JDBC statements;  17894551612 nanoseconds spent executing 20000 JDBC statements;  488607094 nanoseconds spent executing 334 JDBC batches;  B30 – 20K S  156952830 nanoseconds spent preparing 20001 JDBC statements;  20448644616 nanoseconds spent executing 20000 JDBC statements;  921923248 nanoseconds spent executing 667 JDBC batches;  + update  156701324 nanoseconds spent preparing 20002 JDBC statements;  14528873928 nanoseconds spent executing 20000 JDBC statements;  1196814020 nanoseconds spent executing 1334 JDBC batches;  B0 – 20K S  282297229 nanoseconds spent preparing 40000 JDBC statements;  27320296876 nanoseconds spent executing 40000 JDBC statements;  0 nanoseconds spent executing 0 JDBC batches;  + update  281875789 nanoseconds spent preparing 60000 JDBC statements;  60375466751 nanoseconds spent executing 60000 JDBC statements;  0 nanoseconds spent executing 0 JDBC batches;  I i tak są szybsze od S, ale to nie sprawa batch. Gdy jednaj będą wykonaywane update to sytuacja zacznie się zmianiać  B30 – 20K I  329892992 nanoseconds spent preparing 20000 JDBC statements;  18454414179 nanoseconds spent executing 20000 JDBC statements;  0 nanoseconds spent executing 0 JDBC batches;  Jednak dla UPDATE zaczyna się to mianiać  B0 – 20K I + update  179980301 nanoseconds spent preparing 20001 JDBC statements;  22763630378 nanoseconds spent executing 20000 JDBC statements;  933473879 nanoseconds spent executing 667 JDBC batches;  B30 – 20K I + update  271615639 nanoseconds spent preparing 40000 JDBC statements;  41015160239 nanoseconds spent executing 40000 JDBC statements;  0 nanoseconds spent executing 0 JDBC batches; |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |