Hibernate

Identyfikatory, klucze główne, sekwencje

Identyfikatory

Identyfikatory odwzorowują klucze główne tabel po stronie encji. JPA (a więc też Hibernate) zakłada, że klucze są *UNIQUE*, *NOT NULL*. Powinno być także *IMMUTABLE* (JPA nie definiuje tej kwestii). Jeśli ma być zmienną wartością, należy użyć @NaturalId. Teoretycznie identyfikator !=klucz główny, wystarczy żeby spełniał powyższe wymagania. Generalnie jednak używamy terminów identyfikator i klucz główny jako wymienne.

Identyfikatory

@Id

Prosty identyfikator dla pojedynczej kolumny. Wg JPA mogą nim być:

- jakikolwiek typ prosty Javy
- jakakolwiek owijka na typ prosty (Integer, Short, etc)
- java.lang.String
- java.util.Date (TemporalType#DATE)
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

Nie może być wykorzystywany ENUM.

Identyfikatory - pojedynczy identyfikator

@Id

Wartości identyfikatora możemy przypisywać własnoręcznie:

```
@Id
private Long id
```

albo mieć generowane automatycznie (tylko dla short, int, long i owijek!) :

```
@Id
@GeneratedValue
private Long id;
```

Identyfikatory - złożony identyfikator

Na polach złożonych identyfikatorów nie można stosować automatycznego generowania wartości poprzez adnotacje @Generated, @CreationTimestamp, @ValueGenerationType.

Złożone identyfikatory wg JPA muszą mieć owe “klasy klucza głównego”, do których odnosimy się poprzez @IdClass lub @EmbeddedId. Dla Hibernate wystarczy tylko umieścić kilka razy @Id nad wybranymi polami.

Identyfikatory - złożony identyfikator

@IdClass

Zamiast jednego pola możemy mieć klucz główny złożony. Definiujemy wtedy publiczną *klasę klucza głównego* i opatrujemy adnotację @IdClass. Musi mieć publiczny, bezargumentowy konstruktor i być serializowana, oraz equals() i hashCode(). W identyfikatorach złożonych można używać złączeń @ManyToOne - jedno z pól identyfikatora jest kolejną encją, tabelą (nie opisane w prezentacji, patrz dokumentacja).

Identyfikatory - złożony identyfikator

@IdClass

```
@Entity
@IdClass( PK.class)
class Home {
@Id
private String color;
@Id
private Integer number;

class PK implements Serializable{
private String color;
private Integer number;
```

Identyfikatory - złożony identyfikator

@EmbeddedId + @Embeddable

Adnotacje te pozwalają nam na umieszczenie jednego pola w encji, które jest inną klasą, a które jest złożonym identyfikatorem. Klasę, która jest identyfikatorem, oznaczamy jako @Embeddable.

Identyfikatory - złożony identyfikator

@EmbeddedId + @Embeddable

```
@Entity
class Home {
@EmbeddedId
private PK pk;


@Embeddable
class PK implements Serializable {
private String color;
private Integer number;
```

Identyfikatory - złożony identyfikator poprzez relacje (associacje, associactions)

@ManyToOne

```
@Entity(name = "Book")
public class Book implements Serializable {

    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    private Author author;

    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    private Publisher publisher;
```

Identyfikatory - złożony identyfikator poprzez relacje (associacje, associactions)

@ManyToOne

```
@Entity(name = "Author")
public class Author implements Serializable {

    @Id
    private String name;
```

Generatory sekwencji

@GeneratedValue

Generator sekwencji to mechanizm generujący automatycznie wartości dla pola, będącego identyfikatorem.
Wg JPA możemy w tej sytuacji zastosować tylko tylko integer. Hibernate wspiera różne typy.

Generatory sekwencji

@GeneratedValue

strategy:

- GenerationType.AUTO - dostawca JPA sam wybiera najlepszą strategię
- GenerationType.IDENTITY - kolumna identity w tabeli będzie źródłem wartości klucza
- GenerationType.SEQUENCE -w bazie definiowana jest sekwencja, która dostarcza wartości dla klucza
- GenerationType.TABLE - w bazie definiowana jest tabela, w której definiowane są parametry, jak generować wartości klucza

Generatory sekwencji

@GeneratedValue + @SequenceGenerator

GenerationType.*SEQUENCE*

Bazy wspierające:

- Oracle
- SQL Server 2012
- PostgreSQL
- DB2
- HSQLDB

Hibernate używa SequenceStyleGeneratora. Jeśli przełączymy się na bazę, która nie wspiera typu *sequence*, automatycznie przełącza się na *table*.

Generatory sekwencji

@GeneratedValue + @SequenceGenerator

GenerationType.*SEQUENCE*

Sekwencja to obiekt bazodanowy, który na każde żądanie zwraca inkrementowany intiger.
Jest dużo bardziej uniwersalny niż *identity*, ponieważ:

Sekwencja jest niezależna od tabeli i może być wykorzystywana w kilku kolumnach bądź tabelach
Sekwencja może mieć zaalokowane wartości, aby polepszyć wydajność
Sekwencja może definiować krok inkrementacji (wykorzystuje to algorytm Hilo)
Sekwencja nie ogranicza Hibernate JDBC batching
Sekwencja nie ogranicza model dziedziczenia Hibernate

Generatory sekwencji

@GeneratedValue + @SequenceGenerator

GenerationType.*SEQUENCE*

```
@GeneratedValue(generator = "mojSuperGenerator", strategy =
GenerationType.SEQUENCE)
@SequenceGenerator(name = "mojSuperGenerator", initialValue =
21, allocationSize = 120, schema = "klient_nasz_pan",
        catalog = "sprzedaż", sequenceName = "mojaSekwencja")
private Long id;
```

Generatory sekwencji


```
@GeneratedValue
GenerationType.IDENTITY
```

Bazy wspierające:

- Oracle 12c
- SQL Server
- MySQL (AUTO_INCREMENT)
- DB2
- HSQLDB

Generatory sekwencji


```
@GeneratedValue
GenerationType.IDENTITY
```

Generator *identity* inkrementuje automatycznie kolumnę typu integer/bigint. Inkrementacja odbywa się poza aktualną transakcją, dlatego przy roll-backu możemy mieć niektóre wartości id pominięte. Proces generowania wartości jest bardzo efektywny - używany jest wewnętrzny, lekki mechanizm blokowania (w przeciwieństwie do cięższego mechanizmu transakcyjnego).

Wada *identity* to brak możliwości przypisania wartości do pól identyfikatora przed wykonaniem INSERT-u, dlatego nie możemy wykorzystywać strategii write-behind caching (JDBC batching) , polegającej na tym, że wszystkie INSERTY przetrzymywane są w cache'u aż do wykonania commita. Tutaj każdy zapis musi iść osobno, po kolei.

Generatory sekwencji


```
@GeneratedValue
GenerationType.IDENTITY
```

Generator *identity* nie może być też używany przy dziedziczeniu *table per class*, ponieważ ponieważ każda z podklas otrzymałaby ten sam identyfikator.

Generatory sekwencji


```
@GeneratedValue
GenerationType.IDENTITY
```

```
@GeneratedValue(generator = "mojSuperGenerator", strategy =
GenerationType.IDENTITY)
private Long id;
```

Generatory sekwencji

@GeneratedValue + @TableGenerator

GenerationType.*TABLE*

Podczas gdy *identity* lub *sequence* nie tworzą dodatkowych transakcji, tak *table* wymaga obowiązkowo ACID dla synchronizacji generowania id przy dostępie wielowątkowym. Jest to możliwe dzięki blokadzie (lock) na poziomie wiersza tabeli. Zwiększa to koszt pozyskania identyfikatora w porównaniu z generatorami *identity* lub *sequence*.

Generatory sekwencji

@GeneratedValue + @TableGenerator

GenerationType.*TABLE*

Table jest obliczana w osobnej transakcji, wymaga więc izolacji (IsolationDelegatemechanism - wspiera zarówno transakcje bezpośrednie JDBC, jak i poprzez serwer aplikacji JTA (Java Transaction API)).
Dla transakcji lokalnej otwierane jest nowe połączenie, co obciąża aktualny mechanizm puli połączeń.
Dla połączeń globalnych wymagane jest zawieszenie obecnej transakcji, po otrzymaniu wartości id aktualna transakcja jest wznawiana. To generuje koszt i obniża wydajnośc aplikacji.
Ponieważ tabela może zawierać dane dla kilku encji, dlatego czasem trzeba podać `pkColumnName = "id" , pkColumnValue = "3"`, czyli która kolumna tabeli jest kluczem głównym dla niej, oraz która wartość id w tej kolumnie służy do generowania wartości identyfikatora dla naszej encji

Generatory sekwencji

@GeneratedValue + @TableGenerator

GenerationType.*TABLE*

```
@GeneratedValue(generator = "mojSuperGenerator", strategy =
GenerationType.TABLE)
@TableGenerator(name = "mojSuperGenerator", initialValue = 21,
      allocationSize = 120, schema = "klient_nasz_pan",
      catalog = "sprzedaż", table = "tabelaGenerator",
pkColumnName = "id" , pkColumnValue = "3")
```