

Automatyczne pobieranie danych w Hibernate

Bolesław Ziobrowski

Artykuł opisuje mechanizm automatycznego pobierania danych w narzędziu Hibernate, problemy z nim związane oraz optymalny sposób jego wykorzystania.

Hibernate jest darmowym narzędziem mapowania obiektowo - relacyjnego, a więc takim, które stara się rozwiązać rozbrat między modelem relacyjnym a obiektywnym (3). Narzędzie to przenosi operacje wykonane na obiektach w aplikacji do bazy danych, ograniczając potrzebę pisania zapytań w języku SQL. Przeglądając dostępne w internecie proste przykłady, a jest ich całkiem sporo, można dojść do wniosku, że wystarczy kilka adnotacji i problem dostępu do danych jest rozwiązany. Wrażenie to jest tylko pozorne i najczęściej mija z chwilą pojawienia się jakiegoś problemu wydajnościowego. Wtedy to okazuje się, że znajomość algorytmów, ścieżek dostępu i fizycznego projektu bazy danych jest niezbędna. Staje się też oczywiste, że Hibernate jest tylko ładnym opakowaniem nałożonym na JDBC. Aby z niego efektywnie korzystać trzeba zatem nie tylko znać warstwy znajdujące się pod nim, ale też jego własne mechanizmy. Jednym z nich jest automatyczne pobieranie obiektów.

Automatyczne pobieranie obiektów (fetch strategy) jest właściwością związków między encjami i może być leniwe bądź gorliwe. Po wczytaniu obiektu z bazy, jeśli asocjacja ma charakter leniwy to w miejsce niewczytanej strony związku tworzony jest obiekt - pośrednik, który wczytuje dane z bazy tylko na żądanie (gdy zostanie wywołana jakaś jego metoda). Jeśli natomiast asocjacja jest gorliwa, Hibernate wczytuje drugą stronę związku od razu. Mechanizm ten, choć wygląda niewinnie, odgrywa dużą rolę w kwestii wydajności aplikacji. Wykorzystywany bez zastanowienia prowadzi najczęściej do poważnych problemów z wydajnością, których przykłady przedstawiam poniżej.

W przykładach korzystam z prostego trzy - encyjowego model aplikacji - bloga, na który składają się klasy: User, Entry oraz Comment.

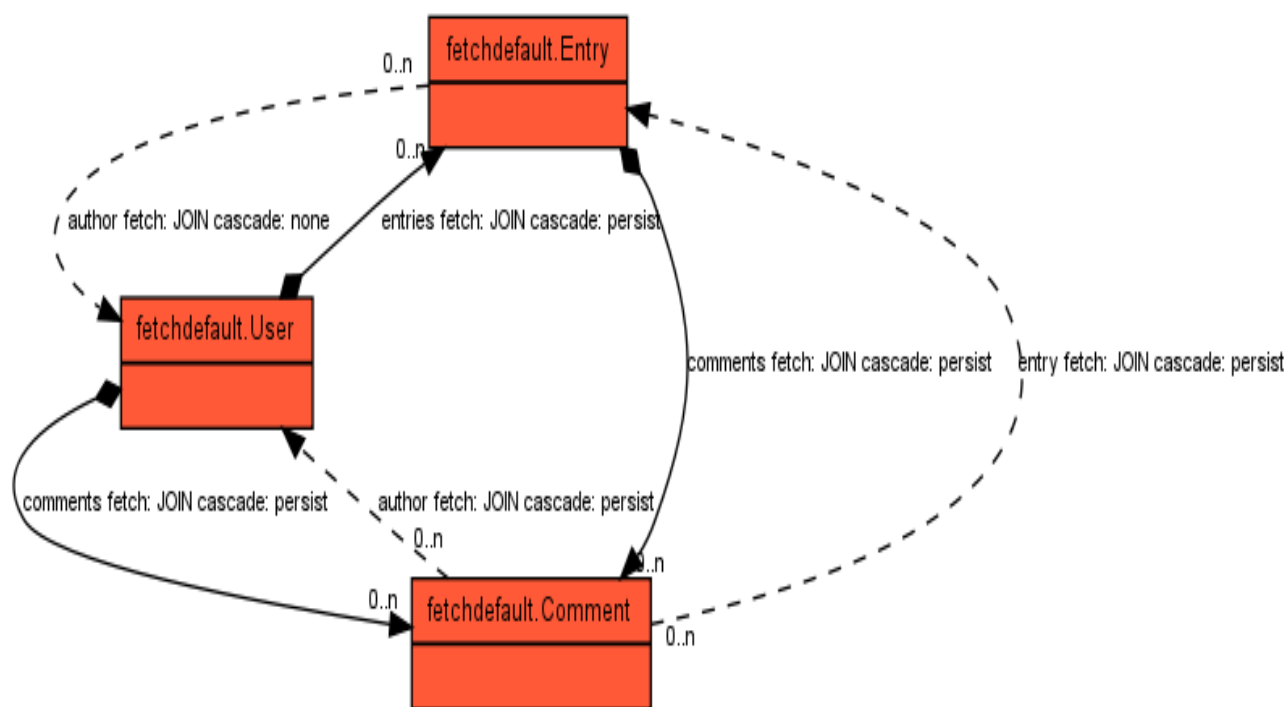


diagram encji wygenerowany z użyciem Linguine Maps (4)

Kod klas:

```
@Entity
@Table(name="customer")
public class User
{
    @Id
    @GeneratedValue
    private Integer id;

    @OneToMany(cascade={javax.persistence.CascadeType.PERSIST}, mappedBy="author")
    private Set<Entry> entries = new HashSet<Entry>();

    @OneToMany(cascade={javax.persistence.CascadeType.PERSIST}, mappedBy="author")
    private Set<Comment> comments = new HashSet<Comment>();

    private String name;

    User(){ /* required by hibernate */ }

    public User( String name )
    {
        this.name = name;
    }

    public void addEntry( Entry entry )
    {
        entries.add( entry );
        entry.setAuthor(this);
    }

    public void addComment( Comment comment )
    {
        comments.add( comment );
        comment.setAuthor(this);
    }

    public Integer getId()
    {
        return id;
    }

    public void walkObjectGraph()
    {
        for ( Entry entry : this.entries )
        {
            entry.walkObjectGraph();
        }
    }

    public Entry writeEntry( String content )
    {
```

```

        return new Entry( this, content );
    }

    public Comment commentEntry( Entry entry, String comment )
    {
        return new Comment( entry, this, comment );
    }
}

```

@Entity

public class Entry

```

{
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne(optional=false)
    private User author;

    @OneToMany(cascade={javax.persistence.CascadeType.PERSIST}, mappedBy="entry")
    private Set<Comment> comments = new HashSet<Comment>();

    private String content;

    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    Entry() { /* required by hibernate */ }

    public Entry( User author, String content )
    {
        this.author = author;
        this.content = content;
        creationDate = new Date();
    }

    public void addComment( Comment comment )
    {
        comments.add( comment );
        comment.setEntry(this);
    }

    public void setAuthor(User author)
    {
        this.author = author;
    }

    public void walkObjectGraph()
    {
        for ( Comment comment : comments )
        {
            comment.walkObjectGraph();
        }
    }
}

```

```

    }
}

@Entity
public class Comment
{
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne(cascade={javax.persistence.CascadeType.PERSIST},optional=false)
    private Entry entry;

    @ManyToOne(cascade={javax.persistence.CascadeType.PERSIST},optional=false)
    private User author;

    private String content;

    Comment() { /* required by hibernate */ }

    public Comment( Entry entry, User author, String content )
    {
        entry.addComment(this);
        author.addComment(this);
        this.content = content;
    }

    public void setAuthor(User author)
    {
        this.author = author;
    }

    public void setEntry(Entry entry)
    {
        this.entry = entry;
    }

    public void walkObjectGraph()
    {
        //
    }
}

```

Słów kilka o konfiguracji przykładów

Testy korzystają z bazy danych Postgresql w wersji 8.4, działającej na tym samym komputerze, co uruchamiane testy.

Dzięki ustawieniu "**hibernate.hbm2ddl.auto**" na wartość „**create**” schemat bazy jest odtwarzany w momencie kreacji SessionFactory, zatem do uruchomienia Przykładów wystarczy utworzyć użytkownika i bazę "test" - odpowiednie polecenia znajdują się w pliku db.sql.

Czasy wykonania operacji podaję wyłącznie po to, by pokazać orientacyjnie jak długo może trwać

wykonanie określonych operacji bazodanowych - liczy się rząd wielkości a nie dokładna wartość. Metody **walkObjectGraph** przechodzą przez hierarchię obiektów w efekcie wywołując ich załadowanie.

Wszystkie przykłady działają według schematu:

1. otwórz transakcję
2. zapisz dane
3. opróżnij zbuforowane w sesji dane
4. wczytaj dane
5. przeglądaj dane
6. wycofaj transakcję (dane nie zostają utrwalone, więc przykład można uruchamiać wiele razy bez obaw o efekty uboczne)

i różnią się zasadniczo tylko sposobem wczytania danych.

Przykład 1

Pobranie encji przy użyciu metody **get** przy domyślnych ustawieniach pobierania obiektów.

```
Session session = HibernateUtility.getSession( persistentClasses );
Transaction transaction = session.beginTransaction();

transaction.begin();

User kowalski = new User("John Smith");
Entry first = kowalski.writeEntry( "Lorem ipsum dolor sit amet, consectetur adipiscing." );
Entry second = kowalski.writeEntry( "Sed ut perspiciatis unde omnis iste natus error sit ." );

kowalski.commentEntry( first, "First!" );
kowalski.commentEntry( first, "unde omnis iste natus error sit " );
kowalski.commentEntry( second, "blablabla" );
kowalski.commentEntry( second, "Level 67 now but things have slowed down on that fro.");

session.persist(kowalski);
session.flush();
session.clear();

long start = System.nanoTime();
User loadedKowalski = (User) session.get(User.class, kowalski.getId());
long middle = System.nanoTime();
loadedKowalski.walkObjectGraph();
long end = System.nanoTime();

transaction.rollback();
session.close();
```

W przypadku tym rekord kowalskiego został wczytany w ciągu 9 ms, całkowity czas wyniósł 39 ms. Zapytania, które Hibernate "pod maską" wykonuje na bazie danych są następujące:

Najpierw wczytywany jest sam rekord użytkownika "kowalski"

```
select
  user0_.id as id0_0_,
```

```

    user0_.name as name0_0_
from
    customer user0_
where
    user0_.id=?

```

potem wpisy, których jest autorem

```

select
    entries0_.author_id as author4_1_,
    entries0_.id as id1_,
    entries0_.id as id1_0_,
    entries0_.author_id as author4_1_0_,
    entries0_.content as content1_0_,
    entries0_.creationDate as creation3_1_0_
from
    Entry entries0_
where
    entries0_.author_id=?

```

następnie wczytuje komentarze wraz z ich autorami dla pierwszego wpisu

```

select
    comments0_.entry_id as entry4_2_,
    comments0_.id as id2_,
    comments0_.id as id2_1_,
    comments0_.author_id as author3_2_1_,
    comments0_.content as content2_1_,
    comments0_.entry_id as entry4_2_1_,
    user1_.id as id0_0_,
    user1_.name as name0_0_
from
    Comment comments0_
left outer join
    customer user1_
        on comments0_.author_id=user1_.id
where
    comments0_.entry_id=?

```

oraz drugiego

```

select
    comments0_.entry_id as entry4_2_,
    comments0_.id as id2_,
    comments0_.id as id2_1_,
    comments0_.author_id as author3_2_1_,
    comments0_.content as content2_1_,
    comments0_.entry_id as entry4_2_1_,
    user1_.id as id0_0_,
    user1_.name as name0_0_
from
    Comment comments0_
left outer join

```

```

customer user1_
    on comments0_.author_id=user1_.id
where
    comments0_.entry_id=?

```

Jak widać komentarze pobierane są łącznie z ich autorami. Wynika to stąd, że pobieranie obiektów dla adnotacji `@ManyToOne` jest domyślnie gorliwe, zaś dla `@OneToMany` - leniwe.

Nie sugerując się odnotowanymi czasami, założmy, że operacja działa zbyt wolno, zatem spróbujemy ją przyspieszyć. Można tego dokonać między innymi zmieniając potrzebne związki z leniwych na gorliwe, a niepotrzebne - na leniwe.

Zmiana ta polega na dodaniu fragmentu `"fetch=FetchType.EAGER"` do adnotacji `ManyToOne` w klasach `User` oraz `Entry`, oraz `fetch=FetchType.LAZY` w klasie `Comment`.

Przykład 2

Pobranie encji przy użyciu metody `get` przy "dostrojonych" ustawieniach pobierania obiektów. Tym razem Hibernate wykonuje zapytanie:

```

select
    user0_.id as id0_3_,
    user0_.name as name0_3_,
    comments1_.author_id as author3_5_,
    comments1_.id as id5_,
    comments1_.id as id2_0_,
    comments1_.author_id as author3_2_0_,
    comments1_.content as content2_0_,
    comments1_.entry_id as entry4_2_0_,
    entries2_.author_id as author4_6_,
    entries2_.id as id6_,
    entries2_.id as id1_1_,
    entries2_.author_id as author4_1_1_,
    entries2_.content as content1_1_,
    entries2_.creationDate as creation3_1_1_,
    comments3_.entry_id as entry4_7_,
    comments3_.id as id7_,
    comments3_.id as id2_2_,
    comments3_.author_id as author3_2_2_,
    comments3_.content as content2_2_,
    comments3_.entry_id as entry4_2_2_
from
    customer user0_
left outer join
    Comment comments1_
        on user0_.id=comments1_.author_id
left outer join
    Entry entries2_
        on user0_.id=entries2_.author_id
left outer join
    Comment comments3_
        on entries2_.id=comments3_.entry_id
where
    user0_.id=?

```

którego wykonanie zajmuje około 30 ms. Nie jest to jakaś znaczna poprawa, ale założmy, że teraz wydajność jest akceptowalna, więc to podejście zostaje zaakceptowane.

Przykład 3

Po dłuższym czasie otrzymujemy zadanie, w którego fragmencie potrzebujemy odczytać dane wybranego użytkownika. Zamiast metody **get**, korzystamy z prostego zapytania w języku HQL,

```
User loadedUser = (User) session.createQuery("from User c where c.id = :id")
    .setParameter("id", user.getId())
    .uniqueResult();
```

co powoduje wywołanie następujących poleceń:

Hibernate wczytuje dane użytkownika tak jak to określa zapytanie

```
select
    user0_.id as id0_,
    user0_.name as name0_
from
    customer user0_
where
    user0_.id=?
```

po czym wczytuje wszystkie jego wpisy,

```
select
    entries0_.author_id as author4_1_,
    entries0_.id as id1_,
    entries0_.id as id1_0_,
    entries0_.author_id as author4_1_0_,
    entries0_.content as content1_0_,
    entries0_.creationDate as creation3_1_0_
from
    Entry entries0_
where
    entries0_.author_id=?
```

ich komentarze,

```
select
    comments0_.entry_id as entry4_1_,
    comments0_.id as id1_,
    comments0_.id as id2_0_,
    comments0_.author_id as author3_2_0_,
    comments0_.content as content2_0_,
    comments0_.entry_id as entry4_2_0_
from
    Comment comments0_
where
    comments0_.entry_id=?
```

```

select
    comments0_.entry_id as entry4_1_,
    comments0_.id as id1_,
    comments0_.id as id2_0_,
    comments0_.author_id as author3_2_0_,
    comments0_.content as content2_0_,
    comments0_.entry_id as entry4_2_0_
from
    Comment comments0_
where
    comments0_.entry_id=?

```

oraz komentarze napisane przez wczytanego użytkownika,

```

select
    comments0_.author_id as author3_1_,
    comments0_.id as id1_,
    comments0_.id as id2_0_,
    comments0_.author_id as author3_2_0_,
    comments0_.content as content2_0_,
    comments0_.entry_id as entry4_2_0_
from
    Comment comments0_
where
    comments0_.author_id=?

```

co zajmuje 210 ms i odbywa się przed wywołaniem metody **walkObjectGraph** .

Jeśli potrzebujemy wyłącznie danych użytkownika to zarejestrowana wydajność jest dużo (20 razy) gorsza od tej z pierwszego przykładu (10 ms). Jest to oczywiście konsekwencja zmiany ustawień ładowania obiektów w przykładzie drugim. Co gorsza, jeśli od czasu realizacji kodu z poprzedniego przykładu minęło trochę czasu, a aplikacja została znacząco rozbudowana, to nie istnieje prosty sposób naprawy sytuacji. Czy pobieramy obiekt metodami **get**, **load**, zapytaniem HQL czy z użyciem Criteria API - Hibernate za każdym razem wczyta dane z gorliwych asocjacji.

Jakie mamy zatem wyjścia ?

1. Jeśli spróbujemy zmienić mapowanie to ryzykujemy wprowadzenie błędów typu `LazyInitializationException` lub pogorszenie wydajności w sekcjach kodu niejawnie polegających na poprzednich ustawieniach.
2. Jeśli wprowadzimy niemapowany jeden obiekt i wczytamy do niego dane za pomocą natywnego zapytania SQL, to tylko komplikujemy naszą aplikację i narażamy się na błędy. Podejście to w istocie nie rozwiązuje problemu a tylko go obchodzi w tym jednym konkretnym przypadku.
3. Jeśli utworzymy nowe mapowanie (`SessionFactory`) to nie dość, że jest to pracochłonne oraz wprowadza podatną na błędy duplikację ustawień (możemy doprowadzić do braku spójności między mapowaniami - na przykład wprowadzić zapytanie do pliku `hbm.xml` tylko dla jednego mapowania), to interakcje między kodem korzystającym z jednego mapowania i drugiego do łatwych nie należą.

Przykład 4

Jest to zasadniczo uzupełnienie poprzedniego przykładu, przy czym tutaj wczytujemy użytkownika wraz z jego wpisami oraz komentarzami. Kod jest następujący:

```
User loadedUser = (User) session.createQuery("from User as user " +
                                           "left outer join fetch user.entries as entry " +
                                           "left outer join fetch user.comments as comment " +
                                           "where user.id = :id")
                                           .setParameter("id", user.getId())
                                           .uniqueResult();
```

Hibernate posłusznie wczytuje wymagane dane

```
select
  user0_.id as id0_0_,
  entries1_.id as id1_1_,
  comments2_.id as id2_2_,
  user0_.name as name0_0_,
  entries1_.author_id as author4_1_1_,
  entries1_.content as content1_1_,
  entries1_.creationDate as creation3_1_1_,
  entries1_.author_id as author4_0__,
  entries1_.id as id0__,
  comments2_.author_id as author3_2_2_,
  comments2_.content as content2_2_,
  comments2_.entry_id as entry4_2_2_,
  comments2_.author_id as author3_1__,
  comments2_.id as id1___
from
  customer user0_
left outer join
  Entry entries1_
  on user0_.id=entries1_.author_id
left outer join
  Comment comments2_
  on user0_.id=comments2_.author_id
where
  user0_.id=?
```

po czym doczytuje komentarze dla wczytanych wpisów,

```
select
  comments0_.entry_id as entry4_1_,
  comments0_.id as id1_,
  comments0_.id as id2_0_,
  comments0_.author_id as author3_2_0_,
  comments0_.content as content2_0_,
  comments0_.entry_id as entry4_2_0_
from
  Comment comments0_
where
  comments0_.entry_id=?

select
```

```

        comments0_.entry_id as entry4_1_,
        comments0_.id as id1_,
        comments0_.id as id2_0_,
        comments0_.author_id as author3_2_0_,
        comments0_.content as content2_0_,
        comments0_.entry_id as entry4_2_0_
from
    Comment comments0_
where
    comments0_.entry_id=?

```

co zajmuje 230 ms. Widać, że niezależnie od tego jaki fragment grafu obiektów chcemy pobrać, reszta i tak zostanie wczytana. Co więcej, komentarze są wczytywane osobno dla każdego wpisu, więc im wydajność spada szybko wraz ze wzrostem ich liczby. Przykładowo - przy dziesięciu wpisach wczytanie danych zajęło 380 ms.

Podsumowanie

Powyższe przypadki wyraźnie pokazują, że korzystanie ze związków gorliwych może być szkodliwe dla wydajności.

Gorliwe związki wiele-do-jednego oraz jeden-do-jednego często nie powodują problemów z wydajnością, gdyż wczytanie jednego niepotrzebnego rekordu nie jest zbyt kosztowne. Istnieją jednak przypadki, w których mogą dać o sobie znać, powodując liczne odwołania do bazy danych. Dla przykładu – gdyby encja Wpis mogła posiadać tylko jeden komentarz (związek byłby typu jeden-do-jednego), to w ostatnim przykładzie każdy komentarz zostałby wczytany osobno, co byłoby bardzo niewydajne.

Innym przypadkiem, na który należy szczególnie uważać wiąże się ze strukturami hierarchicznymi. Jeśli związek węzła z jego rodzicem będzie gorliwy (a domyślnie związki wiele-do-jednego takie są), to jego wczytanie spowoduje pobranie wszystkich jego "przodków" aż do samego korzenia hierarchii. Każdy z tych węzłów – przodków zostanie wczytany osobno, zatem zamiast jednego odwołania bazodanowego, mamy ich tyle, ile wynosi poziom zagłębienia wczytanego rekordu. W trakcie rozwijania aplikacji może to nie być widoczne, gdyż testy są wykonywane na niewielkich hierarchiach o dwóch lub trzech poziomach, więc te dwa dodatkowe odczyty nie pogarszają znacząco wydajności. Z drugiej strony - systemie produkcyjnym hierarchie mogą być całkiem spore, a operacje dużo bardziej kosztowne, co zapewne mocno zmotywuje użytkowników aplikacji do rozbudowywania drzewa wszerz, a nie na wysokość.

O ile związki *-do-jednego w pewnych rzadkich sytuacjach można uznać za stosowne, o tyle asocjacji *-do-wielu najlepiej nigdy nie oznaczać jako gorliwe. Jeśli bowiem to zrobimy, to w każdym przypadku gdy wczytamy dana encje wczytane zostaną również powiązane z nią rekordy. Możliwe, że w obecnie realizowanym przypadku użycia ustawienie to jest przydatne i zaoszczędziłoby trochę czasu, ale czy naprawdę za każdym razem, kiedy wczytamy encje A będziemy potrzebować powiązaną kolekcję encji B ? Co jeśli w kolejnym przypadku użycia dane te będą zbędne ? Co jeśli znacznie spowolni działanie aplikacji ? Pytania tego rodzaju należy postawić sobie za każdym razem, gdy mamy ochotę uczynić związek gorliwym.

Oczywiście w obu przypadkach można spróbować naprawić problem to przez zmianę ustawień w mapowaniu, ale należy pamiętać, że w ten sposób zmieniamy globalną konfigurację Hibernate. Tym samym wpływamy na każdy kod, który z niego korzysta. Przed zrealizowaniem tej zmiany warto więc zadać sobie pytanie - czy przypadkiem nie narażam się w ten sposób na pojawienie się wyjątków **LazyInitializationException** lub problemy z wydajnością ? Jeśli aplikacja jest naprawdę dobrze przetestowana to błędy tego typu *powinny* pojawić się w raporcie testów. Jeśli tak nie jest (a nie niestety jeszcze się z takową nie spotkałem), to koniecznie musimy

przejrzeć każde wykorzystanie danego obiektu lub kolekcji obiektów w naszej aplikacji. Trudność takiego zadania wynika właśnie z niejawnej, ukrytej natury automatycznego pobierania danych. Bez dokładnej analizy kodu nie możemy powiedzieć jakie dane są pobierane z bazy w określonym fragmencie aplikacji.

Dalece lepszym rozwiązaniem jest rezygnacja z pobierania gorliwego i wczytanie potrzebnego zestawu danych za pomocą jednego, bądź wielu, zapytań HQL. Jak widać z przykładów są one bardzo proste do napisania. Postępując w ten sposób uwalniamy się od potencjalnych problemów z wydajnością w przyszłości, czytelnie określając przy tym wykorzystywany przez nas zbiór danych. Zmiana sposobu pobierania danych nie zależy już od globalnych ustawień, więc można ją łatwo i bezpiecznie zmienić, tym bardziej, że wiemy, jakie dane były do tej pory pobierane i zwracane.

Teraz, gdy już wiemy jak kłopotliwe potrafi być gorliwe pobieranie danych, będziemy z niego korzystać bardzo ostrożnie. Niestety postępując w ten sposób nadal możemy wpaść w kłopoty, nieświadomie czyniąc jakiś związek gorliwym. Jak może do tego dojść? Gdyby nieobligatoryjny związek User → Entry był typu jeden-do-jednego, z kluczem obcym w tabeli Entry, to wczytanie obiektu User powodowałoby pobranie drugiej strony tego związku, mimo że użyliśmy `fetch=FetchType.LAZY` w adnotacji `@OneToOne`. Czemu w takim razie ustawienie to nie zostało uszanowane? Aby odpowiedzieć na to pytanie należy wrócić do sposobu, w jaki narzędzie realizuje domyślnie związki leniwe – czyli obiektu pośrednika. Wczytując wyłącznie obiekt User, nie można stwierdzić, czy odpowiedni obiekt Entry istnieje czy – nie wiadomo czy utworzyć obiekt – pośrednika, czy też użyć wartości null. Z tego powodu Hibernate traktuje związek jako gorliwy. Aby ustawienie zadziałało należy przeprowadzić instrumentację kodu bajtowego klasy User – szczegóły znajdują się w (1). Jeśli chcemy uniknąć tej i innych niespodzianek (np. błędów w implementacji narzędzia), musimy to sami sprawdzić - najlepiej w postaci testów jednostkowych. Wbrew pozorom nie jest to skomplikowane i można to zrobić całkiem szybko i łatwo korzystając z metody `Hibernate.isInitialized`.

Testy tego typu mają następujący schemat:

1. Przygotuj i zapisz testowaną encję i wszystkie encje z nią powiązane
2. Wymuś zapis danych do bazy za pomocą `Session.flush()`
3. Wyczyść zbuforowane obiekty wywołując `Session.clear()` (w przeciwnym wypadku w kolejnym kroku Hibernate nie odpyta bazy, tylko zwróci obiekty utworzone w punkcie 1)
4. Wczytaj rekord za pomocą `Session.get()`
5. Na każdej encji i kolekcji encji wywołaj `Hibernate.isInitialized()` i sprawdź, czy wynik jest zgodny z zmierzonym.

Dopiero wtedy można mieć pewność, że automatyczne pobieranie obiektów działa tak, jak chcemy.

Celowo nie omawiam potencjalnych środków, za pomocą których można próbować walczyć z problematycznymi ustawieniami pobierania gorliwego, takich jak grupowanie poleceń (batching), adnotacji `Fetch` i innych. Mimo że są użyteczne, to problemu nie rozwiązują, a wprowadzają przy tym własne efekty uboczne.

Bibliografia:

1. <http://docs.jboss.org/hibernate/core/3.3/reference/en/html/performance.html>
2. <http://www.manning.com/bauer2/>
3. http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
4. <http://www.softwaresecretweapons.com/jspwiki/linguinemaps>