

Testowanie poleceń bazodanowych w Hibernate

Bolesław Ziobrowski

W bieżącym wpisie chciałbym poruszyć problem testowania zapytań realizowanych w oparciu o Hibernate. Wszystkie przykłady wykorzystują encje wykorzystywane w poprzednich artykułach – dla przypomnienia są to:

```
@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 hiberante */ }

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

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

    public Set<Entry> getEntries()
    {
        return entries;
    }

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

    public Integer getId()
    {
        return id;
    }
}

@Entity
public class Entry
{
    @Id
    @GeneratedValue
    private Integer id;
```

```

@ManyToOne(optional=false, cascade={javax.persistence.CascadeType.PERSIST})
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 hiberante */ }

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

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 Set<Comment> getComments()
{
    return this.comments;
}

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

@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( String content )
    {

```

```

        this.content = content;
    }

    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 User getAuthor()
    {
        return this.author;
    }

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

```

Tylko i wyłącznie uproszczenia operacji zapisu wewnątrz testów, wszystkie związki między encjami propagują operację „persist”.

Testy zapytań, których dotyczy artykuł, mają charakter integracyjny, gdyż wykorzystujemy w nich bazę danych, potencjalnie znajdującą się na innym komputerze. Są one potrzebne, by sprawdzić czy zapytanie działa w pożądanym sposobie, jak zachowa się w skrajnych sytuacjach itd. Problemem charakterystycznym testów tego rodzaju jest: „skąd wziąć dane?”. Jeśli dysponujemy lokalną maszyną „integracyjną” na której działa baza z danymi skopiowanymi z systemu produkcyjnego, bądź wygenerowanymi (tak by ich rozkłady, wartości, częstości itp. były zbliżone), to możemy spróbować wyszukać w niej dane, które nadają się do naszego testu i je wykorzystać. Ewentualnie możemy takie dane wpisać samemu w jakimś kliencie bazy danych. Wszystko może działać świetnie do chwili, gdy ktoś owych danych nie zmodyfikuje – przypadkiem, lub celowo – by pasowały do testów jego zapytań. A do tego prędzej czy później dojdzie na pewno. Poza tą niewątpliwą „kruchością” testu, musimy liczyć się z tym, że będzie prawdopodobnie mało czytelny a przez to trudniejszy w utrzymaniu i zrozumieniu w przypadku gdy zakończy się niepowodzeniem. Trudność w odczytaniu testu wynika z tego, że ma on często taką postać:

```

@org.junit.Test
public void testQuery()
{
    List<User> results = query( 101010, "alamakota" );

    Assert.assertEquals( 5, results.size() );
    //...
}

```

Aby zrozumieć co tak naprawdę jest testowane musimy zajrzeć do bazy i sprawdzić dane użytkownika o identyfikatorze 101010 i powiązanych z nim rekordów – a to może zająć dużo czasu, jeśli zapytanie które testujemy jest rozbudowane i sprawdza rekordy przechodząc przez wiele tabel. I teraz wyobraźmy sobie, że nagle okazuje się, że test nie działa, a jego twórca jest na urlopie/ zmienił pracę itp. No dobrze – nawet jeśli to ostatnie nie jest prawdą to i tak jest bardzo wątpliwe by autor kodu pamiętał co w tej chwili natchnienia miał na myśli, a nas czeka mozolne przeglądanie w kliencie bazy danych wyników różnych zapytań nierzadko zawierających setki

kolumn i wierszy – po to tylko by się przekonać, że np. ktoś dodał jedną literę w jakimś polu. Jeśli korzystamy z narzędzia do ciągłej integracji aplikacji (np. Hudsona), to może to cały proces może potrwać jeszcze dłużej, bo będziemy podejrzewać, że to ostatnio zatwierdzona zmiana kodu jest winna całej sytuacji, i dopiero po weryfikacji nowego kodu przejdziemy do szperania w bazie danych. W skrócie – dużo mozolnej pracy i straconego czasu.

Przejdźmy zatem do drugiego rozwiązania, w którym w każdym teście tworzone są dane, na których testujemy zapytanie. Postępując w ten sposób nie musimy już się bać, że jakaś zmiana w bazie zepsuje nasz test, jest on dobrze izolowany od tych zmian. Niestety wiąże się to ze wzrostem złożoności samego testu, który teraz może prezentować się następująco:

```
@org.junit.Test
public void test()
{
    User user = new User( "user" );

    User other1 = new User( "other" );
    Entry entry = new Entry();
    other1.addEntry(entry);

    Comment comment1 = new Comment( "tralala" );
    comment1.setEntry( entry );
    user.addComment(comment1);

    Entry userEntry = new Entry( "bumcykcyk" );
    user.addEntry(userEntry);

    User other2 = new User( "someUser" );
    Comment comment2 = new Comment( "alamakota" );
    comment2.setEntry( userEntry );
    other2.addComment( comment2 );
    //...

    //save all entities
    session.persist(user);
    //...

    List<User> readers = getReadersOf( user, session );

    Assert.assertEquals( 5, results.size() ); ) );
    //...
}
```

Już na powyższym, prostym przykładzie widzimy, że napisanie takiego testu wymaga dużo klepania, a rezultat jest mało czytelny. W rzeczywistej aplikacji utworzenie potrzebnych danych nierzadko wymaga utworzenia kilkudziesięciu obiektów, połączonych w skomplikowany sposób i których pola muszą spełniać różnego rodzaju ograniczenia bazodanowe. To ostatnie potrafi nieźle skomplikować sprawę, w szczególności ograniczenia UNIQUE oraz wyzwalacze – w zamyśle test powinien działać niezależnie od zastanej zawartości bazy danych. Gdy już utworzymy niezbędne obiekty, przychodzi kolej na ich utrwalenie. W przedstawionych testach załatwia to ustawienie propagacji operacji „persist” w każdym związku encji – w rzeczywistej aplikacji tak wygodnie z reguły nie jest, więc musimy sprawdzić reguły propagacji w mapowaniu i utrwalić obiekty w odpowiedniej kolejności. Gdy mamy wiele encji proces ten może zająć trochę czasu – co prawda nie tak dużo jak kreowanie danych, ale wcale nie jest to takie hop-siup. W rezultacie powstaje test liczący kilkadziesiąt lub kilkaset linii kodu, mało czytelny i trudny w diagnozie, utrzymaniu. Cechy te powodują, że wielu programistów uważa pisanie takiego testu za stratę czasu i zwyczajnie tego nie robi.

Wymienione wyżej wady testów tego rodzaju można jednak usunąć: znacznie zmniejszyć

objętość oraz czas potrzebny na pisanie testu oraz zwiększyć jego czytelność. Aby to zrobić musimy w pierwszej kolejności zaobserwować dwie charakterystyczne cechy testów tego rodzaju. Po pierwsze – często w całej aplikacji lub pewnym jej module istnieje swego rodzaju „centralna” encja – węzeł początkowy grafu obiektów, dla której reszta encji stanowi zbiór właściwości, rozszerzeń. Po drugie – najczęściej zapisujemy cały graf obiektów – czyli wszystkie obiekty, które możemy osiągnąć przechodząc od owej encji do jej „właściwości”. W oparciu o te spostrzeżenia uprościmy proces tworzenia testu zapytań w Hibernate.

Gdy w aplikacji nad którą pracujemy istnieje „centralna” encja, to z reguły w testach zapytań najpierw musimy stworzyć właśnie obiekt tego typu, a dopiero potem owe właściwości, dodatkowe informacje, a skoro tak to tworzenie grafu obiektów możemy napisać (w możliwie konfigurowalny sposób) raz i opakować we wzorzec projektowy Budowniczy (ang. *Builder*) i wykorzystywać w każdym z testów. Dla opisanych wcześniej klas – encji, zakładając że to klasa `User` jest „najważniejsza”, rozwiązanie to może wyglądać następująco (metody `createXXX` tworzą obiekty odpowiedniego typu z jakimiś domyślnymi wartościami):

```
public class UserBuilder
{
    private boolean withComment;
    private boolean withEntry;
    private boolean withEntryComment;

    public UserBuilder()
    {
    }

    public UserBuilder withComment()
    {
        this.withComment = true;
        return this;
    }

    public UserBuilder withEntry()
    {
        this.withEntry = true;
        return this;
    }

    public UserBuilder withEntryComments()
    {
        this.withEntryComment = true;
        return this;
    }

    public User build()
    {
        verify();

        User user = createUser( "user" );
        if ( withComment )
        {
            User other = createUser( "other" );
            Entry entry = createEntry();
            other.addEntry(entry);

            Comment comment = createComment( entry, user );
            user.addComment(comment);
        }
        if ( withEntry )
    }
}
```

```

    {
        Entry entry = createEntry();
        user.addEntry(entry);

        if ( withEntryComment )
        {
            User other = createUser( "someUser" );
            createComment( entry, other );
        }
    }

    return user;
}

private void verify()
{
    if ( withEntryComment && !withEntry )
    {
        throw new IllegalStateException( "Entry comment required but
no entry available." );
    }
}
//...
}

```

Teraz nasz test staje się krótszy i dużo bardziej czytelny:

```

@org.junit.Test
public void simplifiedTest()
{
    User user = new UserBuilder()
        .withComment()
        .withEntry()
        .withEntryComments()
        .build();

    //save all entities
    session.persist(user);
    //...

    List<User> readers = getReadersOf( user, session );

    Assert.assertEquals( 5, results.size() ); ) );
    //...
}

```

Oczywiście nic nie stoi na przeszkodzie, by klasa przygotowująca graf obiektów była bardziej skomplikowana – np. pozwalała na tworzenie różnej liczby komentarzy, określała jakieś ważne pola klas itp. Gdy w przyszłości asercje testu nie zostaną spełnione to dużo łatwiej będzie zrozumieć sam test i zlokalizować przyczynę błędu.

Rozwiązanie drugiego z problemów tworzenia testów zapytań, a mianowicie – utrwalania grafu obiektów osiągalnych od obiektu „centralnego” (ang. *persistence by reachability*) wymaga zaimplementowania odpowiedniej procedury. Jest to dużo trudniejsze niż tworzenie klasy budowniczej i może trochę spowolnić testy, ale gdy już to zrobimy nie będziemy musieli sprawdzać za każdym razem ustawień propagacji operacji w mapowaniu, a w praktyce czas narzut takiej metody jest znikomy. Treść takiej metody zostanie omówiona i możliwa do pobrania ze strony w jednym z przyszłych artykułów – kiedyś już napisałem coś takiego w trochę uproszczonym wydaniu, ale gdzieś mi się zapodziało, więc muszę to zrobić ponownie. Stosując metodę utrwalającą graf obiektów test zostaje uproszczony do postaci:

```

@org.junit.Test
public void simplifiedTest()
{
    User user = new UserBuilder()
                .withComment()
                .withEntry()
                .withEntryComments()
                .build();

    //save all entities
    save( user );

    List<User> readers = getReadersOf( user, session );

    Assert.assertEquals( 5, results.size() ); ) );
    //...
}

```

Zamiast pogmatwanego, wielo(set)linijkowego behemota, otrzymujemy prostą i bardzo czytelną metodą, która można bardzo szybko napisać. Aby to osiągnąć musimy jednak wpierw utworzyć odpowiednią klasę budującą oraz metodę utrwalającą oraz dobrze je przetestować.

Korzystając z budowniczego musimy mieć świadomość, że od domyślnych wartości pól obiektów może potencjalnie zależeć wiele testów, więc raz ustalone wartości powinny być rzadko zmieniane. Najlepiej w każdym teście samemu ustawiać wartości kluczowych pól wykorzystywanych w poleceniu bazodanowym, przez co uniezależniamy się od domyślnych wartości i zwiększamy czytelność testu – nie musimy wtedy zaglądać do implementacji budowniczego by poznać owe wartości. Jeśli będziemy postępować w ten sposób to możemy bez problemu zmieniać domyślne wartości, co może okazać się przydatne podczas refaktoryzacji bazy danych – zmiany definicji pól, długości itd., bo wystarczy to zrobić w jednym miejscu dla wszystkich testów.

Omawiając budowniczego warto zahaczyć o zagadnienie niezależne od zaproponowanych w artykule rozwiązań, a mianowicie o to, jakimi danymi testować polecenie. Tworząc klasę–budowniczego, możemy co prawda przyjąć jakie zechcemy wartości domyślne pól - byle tylko spełniały ograniczenia zdefiniowane w bazie danych, ale jeśli postąpimy w ten sposób to będziemy wiedzieć jak polecenie zadziała na systemie produkcyjnym. Jeśli aplikacja została już zainstalowana na takowym systemie, to należy z tego skorzystać i sprawdzić charakterystyki wartości co ważniejszych kolumn w bazie i dopiero wtedy określić wartości domyślne. Przy okazji warto zbudować słownik zawierający po kilka wartości skrajnych i przeciętnych kolumn, tak by programiści mogli je z niego pobierać i ustawiać w miejsce kluczowych dla zapytania pól. Testowanie zapytania wydaje się proste, ale lepiej się upewnić co do tego, na jakie dane może ono trafić – zwłaszcza gdy (broń Boże) istnieją rozbieżności między lokalną i produkcyjną bazą danych. Gdy systemu produkcyjnego jeszcze nie ma, to w oparciu o schemat bazy danych musimy sami zgadywać, na jakie wartości polecenie może się w przyszłości nadziać – pamiętając o prawach Murphy'ego dla baz danych: **jeśli na jakąś kolumnę lub grupę kolumn w bazie danych nie zostało nałożone ograniczenie, to może/mogą przyjąć najgorsze dla zapytania wartości.**

Tak na marginesie to gdy już wystarczająco zweryfikujemy za pomocą testów poprawność polecenia a posiadamy w bazie dane możliwie zbliżone do tych z systemu produkcyjnego, to warto włączyć w Hibernate opcje *show_sql*, skopiować polecenie do klienta bazy danych (np. *sqldeveloper* firmy Oracle) i sprawdzić plan zapytania – czasami można w ten sposób szybko odkryć problemy wydajnościowe wynikające z błędów w Hibernate lub nawet w konfiguracji bazy danych (stwierdzono metodą empiryczną).

Podsumowując – pisanie testów poleceń bazodanowych w oparciu o Hibernate nie musi być pracochłonne, zrudne, monotonne, a wyniki – nieczytelne i trudne w utrzymaniu. Stosując opisane wyżej metody można pisać dobre, niezależne stanu bazy testy szybko i łatwo, a gdy w przyszłości zasygnalizują błąd – to szybko go zidentyfikujemy.