Technologia i rozwiązania

Hibernate Search
Skuteczne wyszukiwanie

Skuteczne wyszukiwanie danych!

Steve Perkins

Helion

PACKT PUBLISHING

Tytuł oryginału: Hibernate Search by Example

Tłumaczenie: Andrzej Bobak

ISBN: 978-83-246-8600-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/hibers
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/hibers.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 3

Wykonywanie zapytań

W poprzednim rozdziale stworzyłeś kilka typów trwałych obiektów i zmapowałeś je do indeksów Lucene na parę różnych sposobów. Do tej pory używałeś jednego rodzaju wyszukiwania na podstawie słowa kluczowego we wszystkich wersjach przykładowej aplikacji.

W tym rozdziale poznasz inne typy wyszukiwań dostępne w DSL Hibernate Search. Dowiesz się również, na czym polegają niektóre istotne operacje, takie jak sortowanie i stronicowanie wyników.

API do mapowania kontra API do tworzenia zapytań

Do tej pory omawialiśmy alternatywne API do mapowania klas na tabele bazy danych za pomocą mapera obiektowo-relacyjnego Hibernate. Możesz mapować swoje klasy zarówno przy użyciu plików XML, jak i adnotacji Hibernate lub JPA. Hibernate Search będzie poprawnie obsługiwać każde ze wspomnianych podejść, o ile uświadomisz sobie drobne różnice między nimi.

Gdy chcesz sprecyzować, z którego API korzysta aplikacja Hibernate, musisz ustalić dwie kwestie. Po pierwsze, istnieje kilka sposobów mapowania klas na tabele w bazie danych. Po drugie, dostępne są różne metody odpytywania bazy danych w czasie pracy programu. Maper obiektowo-relacyjny Hibernate ma tradycyjne API opierające się na klasach `SessionFactory` i `Session` oraz API zgodne ze standardami JPA, bazujące na klasach `EntityManagerFactory` i `EntityManager`.

Myślę, że zauważyłeś, iż w naszym przykładowym kodzie klasy były mapowane na tabele bazodanowe za pomocą adnotacji JPA, natomiast zapytania wykonywano z użyciem obiektu klasy `Session`, pochodzącego z tradycyjnego Hibernate. Na pierwszy rzut oka pewnie wyda Ci się to nieco mylące, jednak API do mapowania oraz API do tworzenia zapytań mogą być stosowane zamiennie.

Pewnie zastanawiasz się, którego z podejść powinieneś użyć w projektach opierających się na Hibernate Search? Z jednej strony trzymanie się standardów ma pewne zalety. Gdy nabierzesz doświadczenia w używaniu JPA, będziesz w stanie wykorzystywać zdobyte umiejętności w projektach posługujących się różnymi implementacjami JPA.

Z drugiej strony API mapera obiektowo-relacyjnego Hibernate jest bardziej rozbudowane niż standard JPA. Ponadto Hibernate Search stanowi rozwinięcie projektu ORM Hibernate. Nie ma również gwarancji, że zmiana implementacji JPA będzie wykonalna bez większych ingerencji w napisane już zapytania.

> Używanie standardu JPA jest, ogólnie rzecz biorąc, wskazane. Jednak maper obiektowo-relacyjny Hibernate okazuje się niezbędny do działania Hibernate Search. W związku z tym nie ma sensu unikać za wszelką cenę rozwiązań z klasycznego Hibernate. Większość kodu źródłowego w tej książce używa adnotacji JPA do mapowania encji oraz klasy `Session`, pochodzącej z tradycyjnego wrappera, do wykonywania zapytań.

Tworzenie zapytań w JPA

Pomimo że skupisz się na wykonywaniu zapytań z użyciem tradycyjnego API, dostępny do ściągnięcia kod źródłowy znajduje się z katalogu *rozdzial3-entitymanager*. Zawiera on alternatywną wersję przykładowej aplikacji, w której zaprezentowano użycie JPA zarówno do mapowania, jak i tworzenia zapytań.

Główna zmiana jest widoczna w servlecie implementującym kontroler odpowiedzialny za wyszukiwanie. Zamiast użyć znanego z Hibernate obiektu typu `SessionFactory` w celu utworzenia obiektu klasy `Session`, skorzystasz z pochodzącego z JPA obiektu klasy `EntityManagerFactory`, aby utworzyć obiekt typu `EntytyManager`:

```
...
// Identyfikator "com.packtpub.hibernatesearch.jpa" jest zadeklarowany
// w "META-INF/persistence.xml."
EntityManagerFactory entityManagerFactory =
   Persistence.createEntityManagerFactory(
   "com.packtpub.hibernatesearch.jpa");
EntityManager entityManager =
   entityManagerFactory.createEntityManager();
...
```

Miałeś już do czynienia z fragmentami kodu prezentującymi zapytania w tradycyjnym API. W dotychczasowych przykładach pochodzące z mapera obiektowo-relacyjnego Hibernate obiekty typu `Session` były opakowane obiektami typu `FullTextSession` wywodzącymi się z Hibernate Search. Te z kolei zwracały obiekty typu `FullTextQuery`, które implementowały interfejs `org.hibernate.Query`:

```
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
...
org.hibernate.search.FullTextQuery hibernateQuery =
   fullTextSession.createFullTextQuery(luceneQuery, App.class);
...
```

Zauważ, że obiekty typu `FullTextQuery`, implementujące interfejs `javax.persistence.Query`, tworzą obiekty typu `FullTextEntityManager`, natomiast pochodzące z JPA obiekty `EntityManager` są w nie opakowane:

```
...
FullTextEntityManager fullTextEntityManager =
      org.hibernate.search.jpa.Search.getFullTextEntityManager(
      entityManager);
...
org.hibernate.search.jpa.FullTextQuery jpaQuery =
      fullTextEntityManager.createFullTextQuery(luceneQuery, App.
      class);
...
```

Pochodząca z tradycyjnego API Hibernate klasa `FullTextQuery` jest bardzo podobna do jej odpowiednika wywodzącego się z JPA. Obie zapewniają dostęp zarówno do poznanych już funkcjonalności Hibernate Search, jak i tych, które zostaną przedstawione w dalszych częściach książki. W zasadzie jedyną istotną różnicą jest to, że pochodzą z różnych pakietów.

> Każdy obiekt typu `FullTextQuery` może być rzutowany z powrotem na typ zapytania, z którego został utworzony. Taka operacja wymaga bezpośredniego dostępu do metod Hibernate Search, w związku z tym należy ją wykonywać po skończeniu pracy z obiektem.
>
> Jeżeli mimo wszystko potrzebujesz dostępu do niestandardowych metod po rzutowaniu na zapytanie JPA, możesz użyć metody `unwrap()`, która pozwala na dostęp do implementacji `FullTextQuery`.

Konfiguracja projektu dla Hibernate Search i JPA

Gdy nasz mavenowy projekt zawiera zależność `hibernate-search`, automatycznie w tle dołączonych zostaje około czterdziestu zależności. Niestety, żadna z nich nie zapewnia tworzenia zapytań JPA. Aby mieć dostęp do tej funkcjonalności, musisz samodzielnie dodać zależność `hibernate-entitymanager`.

Jej wersja powinna być zgoda z wersją dodanej wcześniej zależności `hibernate-core`, natomiast niewykluczone, że nie będzie zgodna z wersją `hibernate-search`.

Jeżeli Twoje środowisko programistyczne nie dysponuje funkcjonalnością wizualizowania hierarchii zależności, zawsze możesz użyć komendy Mavena:

```
mvn dependency:tree
```

```
[INFO] com.packpub.hibernatesearch.rozdzial3:rozdzial3:war:0.0.1-SNAPSHOT
[INFO] +- org.hibernate:hibernate-search:jar:4.2.0.Final:compile
[INFO] |  \- org.hibernate:hibernate-search-orm:jar:4.2.0.Final:compile
[INFO] |     +- org.hibernate:hibernate-search-engine:jar:4.2.0.Final:compile
[INFO] |     |  +- org.hibernate.common:hibernate-commons-annotations:jar:4.0.1.
Final:compile
[INFO] |     |  +- org.apache.lucene:lucene-core:jar:3.6.2:compile
[INFO] |     |  +- org.jboss.logging:jboss-logging:jar:3.1.0.GA:compile
[INFO] |     |  +- org.apache.avro:avro:jar:1.6.3:compile
[INFO] |     |  |  +- org.codehaus.jackson:jackson-core-asl:jar:1.8.8:compile
[INFO] |     |  |  +- org.codehaus.jackson:jackson-mapper-asl:jar:1.8.8:compile
[INFO] |     |  |  +- com.thoughtworks.paranamer:paranamer:jar:2.3:compile
[INFO] |     |  |  \- org.xerial.snappy:snappy-java:jar:1.0.4.1:compile
[INFO] |     |  \- org.apache.lucene:lucene-facet:jar:3.6.2:compile
[INFO] |     +- org.hibernate:hibernate-core:jar:4.1.9.Final:compile
```

Jak widać na powyższym rysunku, Hibernate Search w wersji 4.2.0 Final używa mapera obiektowo-relacyjnego Hibernate w wersji 4.1.9 Final. W związku z tym zależność `hibernate-entitymanager` również powinna być dodana w wersji 4.1.9 Final:

```
...
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>4.1.9.Final</version>
</dependency>
...
```

Hibernate Search DSL

W rozdziale 1. — „Twoja pierwsza aplikacja" — zapoznałeś się z Hibernate Search DSL, najwygodniejszym narzędziem do pisania zapytań. Za pomocą DSL łączysz wywołania metod w łańcuch przypominający składnią specjalistyczny język programowania. Jeżeli pisałeś już zapytania z użyciem kryteriów w maperze obiektowo-relacyjnym Hibernate, podejście z zastosowaniem omawianego narzędzia wyda Ci się znajome.

Niezależnie od tego, czy używasz tradycyjnego obiektu `FullTextSession`, czy znanego z JPA obiektu `FullTextEntityManager`, każdy z nich przekazuje do Lucene zapytanie utworzone za pomocą klasy `QueryBuilder`. Jest ona początkowym elementem Hibernate Search DSL, udostępniającym kilka typów zapytań Lucene.

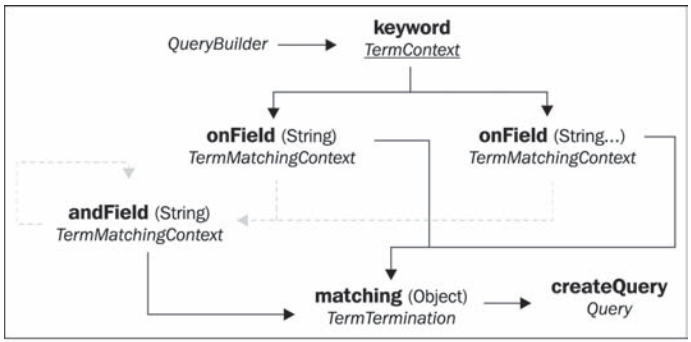
Zapytania na podstawie słów kluczowych

Najprostszym rodzajem zapytania, z którym, nawiasem mówiąc, zdążyłeś się już zetknąć, jest to utworzone na podstawie **słów kluczowych**. Jak sama nazwa wskazuje, wyszukuje ono obiekty zawierające wskazane słowa kluczowe.

Tworzenie zapytania rozpoczyna się od uzyskania obiektu typu `QueryBuilder`, skonfigurowanego dla zadeklarowanej klasy encji:

```
...
QueryBuilder queryBuilder =
   fullTextSession.getSearchFactory().buildQueryBuilder()
      .forEntity(App.class ).get();
...
```

Kolejne kroki zostały przedstawione na poniższym diagramie. Szare przerywane linie przedstawiają opcjonalne kroki:



Tworzenie zapytania na podstawie słów kluczowych
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Kod źródłowy Javy prezentujący zapytanie na podstawie słów kluczowych z użyciem DSL wygląda mniej więcej tak:

```
...
org.apache.lucene.search.Query luceneQuery =
   queryBuilder
   .keyword()
   .onFields("name", "description", "supportedDevices.name",
      "customerReviews.comments")
   .matching(searchString)
   .createQuery();
...
```

Parametr przekazywany do metody `onField` to nazwa indeksowanego pola encji. Jeżeli pole nie jest składową indeksu Lucene, zapytanie będzie błędne. Pola powiązanych encji lub wbudowanych obiektów mogą być również przeszukiwane. W tym celu użyj formatu [*nazwa pola złożonego*].[*nazwa pola w obiekcie złożonym*], np. `supportedDevices.name`.

Możesz dodatkowo posłużyć się metodą `andField`, by uwzględnić kolejne pola w wyszukiwaniu. Jej parametrem jest, podobnie jak w przypadku `onField`, indeksowane pole encji. Ewentualnie możesz użyć metody `onFields`, by jednocześnie zadeklarować kilka pól, tak jak w przedstawionym powyżej przykładzie.

Metoda porównująca pobiera słowa kluczowe, na podstawie których ma się odbyć wyszukiwanie. Zazwyczaj przekazywany zostaje ciąg znaków, jednak technicznie jest możliwe przekazanie dowolnego obiektu, w razie gdyby wymagane było konwertowanie wartości pola (przetwarzaniem pól zajmiemy się szerzej w następnym rozdziale). Zakładając, że przesyłasz słowo lub grupę słów oddzielonych spacjami, Hibernate Search rozbije listę na pojedyncze elementy i wykona wyszukiwanie na podstawie każdego z nich osobno.

Metoda `createQuery` kończy wyrażenie w DSL i zwraca obiekt zapytania Lucene, który następnie może zostać wykorzystany przez obiekt typu `FullTextSession` lub `FullTextEntityManager` w celu utworzenia obiektu typu `FullTextQuery` używanego przez Hibernate Search.

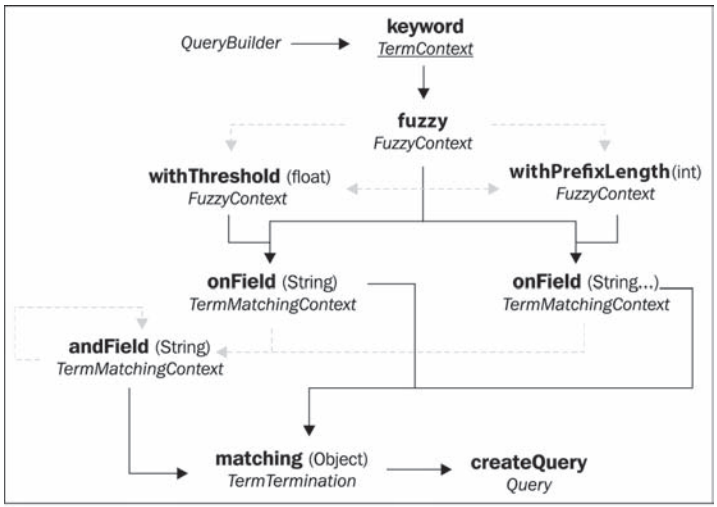
```
...
FullTextQuery hibernateQuery =
   fullTextSession.createFullTextQuery(luceneQuery, App.class);
...
```

Wyszukiwanie rozmyte

Gdy używasz mechanizmu wyszukiwania, zakładasz, że będzie na tyle „sprytny”, by automatycznie poprawić literówki, gdy słowo wprowadzone przez Ciebie jako parametr wyszukiwania jest prawie poprawne. Jeden ze sposobów uzyskania takiego zachowania w Hibernate Search to sprawienie, by zapytania na podstawie słów kluczowych uwzględniały częściowe dopasowanie słów kluczowych.

W przypadku wyszukiwania **rozmytego** wartość pola jest zgodna ze słowem kluczowym, nawet jeśli różni się o jeden lub więcej znaków. Zapytanie ma zdefiniowany **próg zgodności** słowa kluczowego z porównywaną wartością. Próg jest liczbą rzeczywistą w skali od `0` do `1`, gdzie `0` oznacza, że każdy element pasuje do słowa kluczowego, a `1` sygnalizuje, że wyłącznie identyczny element ma być uznany za pasujący. Stopień rozmycia zapytania określasz wartością progu. Im bliżej zera, tym mniej dokładne musi być dopasowanie.

Budowanie zapytania wygląda identycznie jak budowanie zapytania na podstawie słów kluczowych. Jedyną różnicą jest możliwość zadeklarowania, że zapytanie ma być rozmyte, a także zdefiniowania opcjonalnego progu rozmycia. Diagram przedstawiony poniżej przedstawia kolejne kroki budowania zapytania:



Tworzenie zapytania rozmytego
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Metoda `fuzzy` oznacza zapytanie jako rozmyte, z domyślnym progiem o wartości `0,5`. Opcjonalnie za pomocą metody `withThreshold` możesz określić inny poziom rozmycia. Następnie zapytanie jest budowane w sposób przedstawiony już podczas omawiania wyszukiwania z użyciem słów kluczowych. Przykłady do tego rozdziału mają ustawiony próg rozmycia na wartość `0,7`. Dzięki temu unikniesz niewłaściwych dopasowań, a jednocześnie umożliwisz wyszukanie na podstawie słowa, w którym popełniono literówkę, np. błędnie wpisane słowo kluczowe „rodio" zostanie dopasowane do słowa „radio".

```
...
luceneQuery = queryBuilder
   .keyword()
   .fuzzy()
   .withThreshold(0.7f)
   .onFields("name", "description", "supportedDevices.name",
      "customerReviews.comments")
   .matching(searchString)
   .createQuery();
...
```

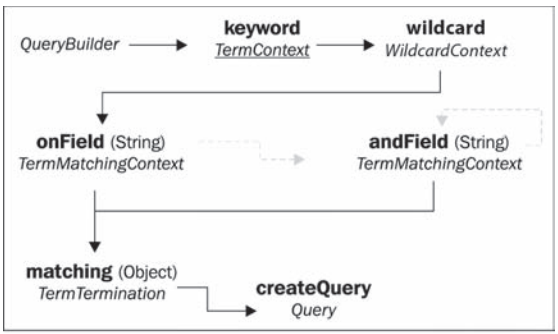
Dodatkowo za pomocą metody `withPrefixLength` możesz zadeklarować, ile początkowych liter w słowach nie powinno zostać rozmyte.

Wyszukiwanie z wieloznacznikami

Drugi z wariantów wyszukiwania z użyciem słów kluczowych nie wymaga stosowania zaawansowanych algorytmów matematycznych. Jeżeli kiedykolwiek posługiwałeś się wzorcem `*.java`, by wyświetlić wszystkie pliki źródłowe Javy w katalogu, to podejście wyda Ci się dosyć znajome.

Wywołanie metody **`wildcard`** na obiekcie typu `QueryBuilder` spowoduje, że słowa kluczowe zawierające znaki `?` oraz `*` będą traktowane niestandardowo. Znak `?` aplikacja zinterpretuje jako substytut jednego dowolnego znaku, np. słowo kluczowe `201?` zostanie dopasowane do wartości `201a`, `2010`, `2011` itp.; `*` stanie się substytutem dowolnego, również pustego, ciągu znaków, np. słowo kluczowe `dom*` będzie dopasowane do wartości `domy`, `domek`, `dom` itp.

Budowa zapytania wygląda identycznie jak budowa zapytania z użyciem słów kluczowych. Jedyną różnicą jest wywołanie bezparametrowej metody `wildcard` na obiekcie typu `QueryBuilder` na początku budowy zapytania.

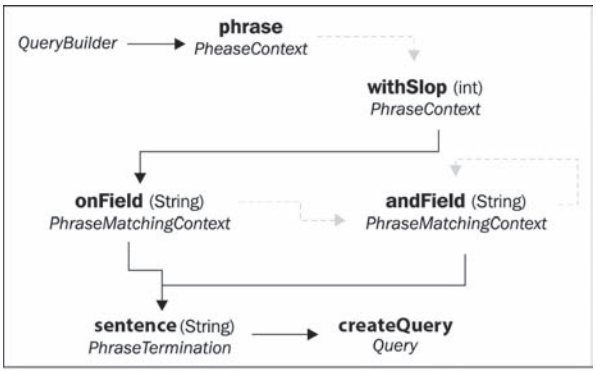


Tworzenie zapytania z wieloznacznikami
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Wyszukiwanie na podstawie dokładnej frazy

Wpisując słowa kluczowe do wyszukiwarki, spodziewasz się, że wyniki wyszukiwania będą zawierały jedno lub więcej z nich. Niekoniecznie wszystkie słowa kluczowe muszą być obecne w każdym z rezultatów. Ich kolejność również nie ma znaczenia.

Przyjęło się, że jeżeli wprowadzisz słowa kluczowe w cudzysłowie, spodziewasz się, iż wyniki wyszukiwania będą zawierać dokładną frazę, którą wpisałeś. DSL Hibernate Search udostępnia **wyszukiwanie na podstawie frazy**.



Tworzenie zapytania na podstawie dokładnej frazy
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Metody `onField` oraz `andField` działają tak jak zapytania na podstawie słów kluczowych. Z kolei `sentence` różni się od `matching` tym, że jej parametrem musi być zmienna typu `String`.

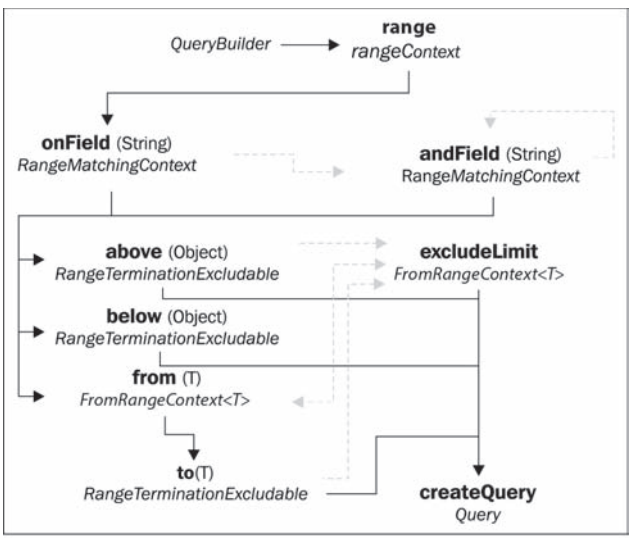
Zapytania na podstawie dokładnej frazy mogą być w ograniczonym stopniu rozmyte. By uzyskać ten efekt, wywołaj na obiekcie typu `QueryBuilder` metodę `withSlop` z liczbą całkowitą jako parametrem określającym, ile dodatkowych słów może się znaleźć w obrębie frazy uznawanej za poprawny wynik.

Jeżeli użytkownik wprowadzi słowa kluczowe w cudzysłowie, w naszej aplikacji zostanie wywołany fragment kodu:

```
...
   luceneQuery = queryBuilder
   .phrase()
   .onField("name")
   .andField("description")
   .andField("supportedDevices.name")
   .andField("customerReviews.comments")
   .sentence(unquotedSearchString)
   .createQuery();
...
```

Zapytania na podstawie zakresu

Zapytania na podstawie fraz lub słów kluczowych polegają na porównywaniu pól z konkretnym wyrażeniem. **Zapytanie oparte na zakresie** jest nieco inne. Sprawdza, czy wartość pola spełnia konkretny warunek, np. czy jest większa lub mniejsza od wskazanej wartości lub czy mieści się we wskazanym zakresie.



Tworzenie zapytania na podstawie zakresu
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Zakresy zazwyczaj tworzy się na podstawie wartości liczbowych lub dat. Niemniej jednak możesz to również zrobić, opierając się na wartości typu `String`. W tym przypadku porównanie będzie się odbywać wg kolejności alfabetycznej.

Metoda `above` sprawdza, czy wartość pola jest wyższa niż wartość przekazana do metody jako parametr. Analogicznie `below` kontroluje, czy wartość pola jest niższa. Natomiast jeżeli chcesz sprawdzić, czy wartość pola mieści się w konkretnym zakresie, zastosuj parę `from` oraz `to` (pamiętaj, że muszą być użyte razem).

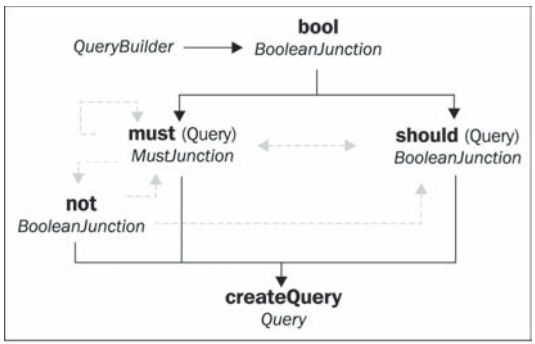
Na każdej z powyższych metod można wywołać bezparametrową metodę `excludeLimit`. Jej zadaniem jest wskazanie, że dany warunek ma być relacją nieostrą. Przykładowo łańcuch `from(5).to(10).excludeLimit()` oznacza zakres `5 <= x < 10`. Metoda `excludeLimit` została wywołana na `to`. Mogła być jednak wywołana na `from` lub obu metodach jednocześnie.

Założyliśmy, że w naszym Bazarze Aplikacji pole `CustomerReview.stars` nie będzie indeksowane. Gdybyś jednak oznaczył je adnotacją `@Field`, w następujący sposób mógłbyś wyszukać aplikacje oznaczone czteroma lub pięcioma gwiazdkami:

```
...
luceneQuery = queryBuilder
   .range()
   .onField("customerReviews.stars")
   .above(3).excludeLimit()
   .createQuery();
...
```

Boolowskie (łączone) zapytania

Załóżmy, że musisz utworzyć złożone zapytanie, którego wyniki powinny spełniać *kilka* warunków umieszczonych w podzapytaniach. Hibernate Search umożliwia łączenie zapytań za pomocą boolowskiej logiki:



Tworzenie zapytania boolowskiego
(szare przerywane strzałki reprezentują opcjonalne ścieżki)

Metoda `bool` wskazuje, że masz do czynienia z zapytaniem kombinacyjnym. Następnie należy użyć co najmniej jednej metody `must` lub `should`. Parametrem każdej z nich jest zapytanie Lucene.

Jeżeli użyjesz `must`, wynikiem wyszukiwania będzie obiekt spełniający warunki podane w zagnieżdżonym zapytaniu. Zastosowanie jednocześnie kilku takich metod odpowiada **logicznemu wyrażeniu `AND`**. Wynikiem wyszukiwania będzie obiekt spełniający wszystkie warunki wszystkich zapytań.

Opcjonalna metoda `not` służy logicznemu zanegowaniu `must`. Oznacza to, że obiekt jest wynikiem wyszukiwania, jeżeli nie spełnia warunków zagnieżdżonego zanegowanego zapytania.

Metoda `should` przypomina **logiczną operację `OR`**. Jeżeli łączone zapytanie składa się wyłącznie z metod `should`, obiekt będący wynikiem zapytania musi spełniać co najmniej jedno z zawartych w nich podzapytań.

> Możesz łączyć metody `must` oraz `should`. W takim przypadku podzapytania w `should` stają się opcjonalne. Jeżeli podzapytania w metodach `must` są spełnione, obiekt zostaje zwrócony jako poprawny wynik zapytania. Nie ma znaczenia, czy spełnia warunki w podzapytaniach zawartych w `should`. Jeżeli warunki dowolnego podzapytania w `must` nie są spełnione, obiekt jest traktowany jako niespełniający warunków całego zapytania. Jeśli używasz jednocześnie obu typów metod, wyniki podzapytań w `should` służą wyłącznie do ustalenia istotności rezultatu wyszukiwania.

Poniższy przykład łączy zapytanie na podstawie słów kluczowych z zapytaniem opartym na zakresie. Poszukiwane są aplikacje zawierające słowo kluczowe „xPhone" oraz mające pięciogwiazdkowe opinie użytkowników:

```
...
luceneQuery = queryBuilder
   .bool()
   .must(
      queryBuilder.keyword().onField("supportedDevices.name")
      .matching("xphone").createQuery()
   )
   .must(
      queryBuilder.range().onField("customerReviews.stars")
      .above(5).createQuery()
   )
   .createQuery();
...
```

Sortowanie

Domyślnie wyniki wyszukiwania zostają zwrócone w kolejności odzwierciedlającej ich trafność. Innymi słowy, są posortowane wg stopnia dopasowania do zapytania. To zagadnienie zostanie omówione w następnych dwóch rozdziałach. Nauczysz się również modyfikować wyliczenia ważności.

Masz jednak możliwość ustalenia kryteriów sortowania. Zazwyczaj chcesz, by wyniki wyszukiwania zostały uporządkowane alfabetycznie, według daty lub wartości liczbowych. Od tej pory we wszystkich wersjach naszego Bazaru Aplikacji użytkownicy będą mogli sortować rezultaty po nazwie aplikacji.

Musisz podjąć pewne kroki, aby umożliwić porządkowanie wyników na podstawie pola mapowanego w indeksie Lucene. Zazwyczaj gdy pole jest indeksowane, domyślny analizator rozbija ciąg znaków na pojedyncze słowa. Jeżeli pole `name` encji typu `App` ma wartość „Sfrustrowane Flamingi", w indeksie Lucene pojawiają się wpisy dla „sfrustrowane" oraz „flamingi". Dzięki temu masz możliwość budowania bardziej wydajnych zapytań. Co jednak, jeśli chcesz sortować wyniki na podstawie oryginalnej, nierozbitej wartości?

Prosty sposób to dwukrotne mapowanie pola. Jak przekonałeś się podczas lektury rozdziału 2. — „Mapowanie klas encji" — w Hibernate Search jest dostępna adnotacja `@Fields`. Możesz użyć jej do opakowania kilku adnotacji `@Field`, każdej z innymi ustawieniami analizatora.

Poniższy fragment kodu prezentuje dwa mapowania jednego pola. Pierwsze jest zadeklarowane z domyślnymi ustawieniami analizatora. Drugie ma nadany parametr `analyze` o wartości `Analyze.NO`, zapobiegający rozbijaniu wartości pola, a także nadaną unikalną nazwę pola w indeksie Lucene:

```
...
@Column
@Fields({
   @Field,
   @Field(name="sorting_name", analyze=Analyze.NO)
})
private String name;
...
```

Nowe pole może zostać użyte do zbudowania obiektu Lucene typu `SortField` i dodania go do obiektu Hibernate Search typu `FullTextQuery`:

```
import org.apache.lucene.search.Sort;
import org.apache.lucene.search.SortField;
...
Sort sort = new Sort(
   new SortField("sorting_name", SortField.STRING));
hibernateQuery.setSort(sort); // obiekt typu FullTextQuery
```

Gdy obiekt `hibernateQuery` zwróci wyniki wyszukiwania, będą one posortowane rosnąco po nazwie aplikacji.

Aby porządkować w kolejności malejącej, wywołaj trójparametrowy konstruktor klasy `SortField`. Trzeci z parametrów, typu `Boolean`, precyzuje, czy sortowanie ma się odbyć rosnąco (wartość `Boolean.FALSE`), czy malejąco (wartość `Boolean.TRUE`).

Stronicowanie

Zazwyczaj gdy wynik zapytania zawiera bardzo dużo elementów, nie jest wskazane przedstawianie ich wszystkich jednocześnie. Popularne rozwiązanie w takiej sytuacji stanowi stronicowanie, czyli prezentowanie wyników porcjami.

Metody dostępne w obiekcie Hibernate Search typu `FullTextQuery` sprawiają, że jest to bardzo proste:

```
...
hibernateQuery.setFirstResult(10);
hibernateQuery.setMaxResults(5);
List<App> apps = hibernateQuery.list();
...
```

Metoda `setMaxResults` wskazuje maksymalną liczbę wyników zwracanych na jednej stronie. W powyższym przykładzie zadeklarowano, że wynik będzie zawierał nie więcej niż pięć elementów, niezależnie od liczby obiektów pasujących do zapytania.

Stronicowanie nie miałoby sensu, gdybyś za każdym razem otrzymywał pierwsze pięć wyników. Powinieneś móc pobierać kolejne strony z wynikami. Używając metody `setFirstResult`, wskazujesz początkowe miejsce.

Przedstawiony powyżej fragment kodu wskazuje, że interesuje Cię pięć wyników, począwszy od jedenastego (wartość parametru wynosi `10`, jednak indeksowanie odbywa się od zera). Aby w kolejnym kroku przetworzyć kolejne pięć wyników, wystarczy, że w powyższym przykładzie zamienisz pierwszą linię na `hibernateQuery.setFirstResult(15)`.

Ostatnią niezbędną informacją jest łączna liczba wyników pasujących do zapytania. Potrzebujesz jej, aby wyliczyć, ile stron z wynikami będziesz prezentować:

```
...
intresultSize = hibernateQuery.getResultSize();
...
```

Metoda `getResultSize` jest bardziej interesująca, niż mogłoby się wydawać na pierwszy rzut oka. Do określenia liczby pasujących rezultatów używa indeksów Lucene. W odróżnieniu od zapytania Lucene tradycyjne zapytanie bazodanowe, wyszukujące wszystkie rezultaty, pochłonęłoby dużo zasobów.

W tym rozdziale nasza aplikacja została wzbogacona o mechanizm stronicowania. Każda strona wyników prezentuje pięć rezultatów wyszukiwania. Przyjrzyj się klasie `SearchServlet` oraz plikowi *search.jsp*, aby zobaczyć, jak wykorzystano w nich informację o łącznej liczbie wyników. Zbadaj również, w jaki sposób zostały w nich użyte dane o łącznej liczbie obiektów spełniających warunki wyszukiwania oraz punkt początkowy wyników w celu zbudowania odnośników „Poprzednie" oraz „Następne".

Nasza aplikacja wygląda obecnie tak:

Podsumowanie

W tym rozdziale przedstawiono najpopularniejsze przypadki użycia zapytań Hibernate Search. Możesz już korzystać z Hibernate Search, niezależnie od tego, czy Twoja aplikacja używa JPA. Poznałeś główne typy zapytań tworzonych za pomocą Hibernate Search DSL. Przyjrzałeś się również diagramom prezentującym sposób budowania poszczególnych zapytań.

Umiesz już sortować rosnąco lub malejąco wyniki zapytania na podstawie dowolnego indeksowanego pola. Potrafisz dzielić je na strony w celu usprawnienia wydajności procesu wyszukiwania oraz przejrzystości prezentacji rezultatów. Funkcjonalność wyszukiwania w Bazarze Aplikacji jest teraz zbliżona do wielu produkcyjnych aplikacji używających Hibernate Search.

W kolejnym rozdziale poznasz zaawansowane techniki mapowania, takie jak niestandardowe typy danych oraz kontrolowanie szczegółów procesu indeksowania w Lucene.

Skorowidz

A

B

D

E

F

H

I

J

K

L

Ł

M

O

P

R

S

T

U

W

Z

Hibernate Search
Skuteczne wyszukiwanie

Użytkownicy aplikacji prawdopodobnie najczęściej korzystają z funkcji wyszukiwania. Ze strony interfejsu użytkownika problem wydaje się trywialny. Zupełnie inne zdanie na ten temat mają programiści. Przygotowanie wydajnego, intuicyjnego i szybkiego mechanizmu wyszukiwania to ogromne wyzwanie. Liczne kryteria, wyszukiwanie tekstu, zależności i połączenia logiczne to tylko część zagadnień, które trzeba opracować. Na szczęście z pomocą przychodzi narzędzie Hibernate Search.

Jest to dodatek do popularnej nie tylko w świecie Javy biblioteki Hibernate. Dzięki Hibernate Search wprowadzenie rozbudowanych mechanizmów wyszukiwania w Twojej aplikacji będzie całkowicie bezbolesne. W trakcie lektury tej książki dowiesz się, jak zamapować encję, jak budować zapytania oraz zarządzać indeksami. Ponadto poznasz zaawansowane strategie poprawy wydajności oraz pełnię możliwości zapytań. Książka ta jest doskonałym podręcznikiem dla wszystkich programistów języka Java, mających za zadanie wprowadzenie do aplikacji rozbudowanych mechanizmów wyszukujących. Hibernate Search to dla nich prawdziwe koło ratunkowe!

Błyskawicznie zaprojektuj rozbudowany mechanizm wyszukiwania dzięki Hibernate Search!

Dzięki tej książce:

- poznasz możliwości biblioteki Hibernate Search
- opanujesz najlepsze strategie zarządzania indeksami
- błyskawicznie zbudujesz zapytanie
- zachwycisz użytkowników Twojej aplikacji rozbudowaną wyszukiwarką

ISBN 978-83-246-8600-1

Cena: 74,90 zł