Enterprise

Een belangrijk onderdeel van veel applicaties is het goed kunnen doorzoeken van data. Gebruikers willen geen ingewikkelde formulieren om hun query samen te stellen, maar willen een Google-like zoekmogelijkheid. Uiteraard moet het zoekresultaat binnen een paar seconden op het scherm staan.

Zoeken in de database zonder database

Hibernate search

Alan van Dam
(alan.van.dam@amis.nl
Jeroen van Wilgenburg
(wilgenburg@amis.nl)

Met het gebruik van SQL is het niet gemakkelijk een generieke zoek-query samen te stellen. Met een *like* en een wildcard (%) kan een hoop bereikt worden, maar het heeft zijn beperkingen. Niet alle kolommen hebben het juiste type om een *like* te gebruiken. Het gebruik van een zoekwoord dat lijkt op het gezochte woord is al helemaal onmogelijk. Dit is handig als er bijvoorbeeld een typefout in de zoekterm zit. Kunnen aangeven hoe belangrijk een bepaalde zoekterm is, wordt ook niet ondersteund door standaard SQL. Moet de zoekterm ook nog in verscheidene kolommen tegelijk voorkomen dan wordt het probleem alleen maar groter. Al met al is het niet eenvoudig om met standaard SQL een goede Google-like zoekfunctie te bouwen.

Full-text search

Door elke rij in de database als een grote string te beschouwen, zou je gemakkelijker met een SQL-query en één like where-clause de juiste rij kunnen vinden. Door alles om te zetten naar tekst kan er gemakkelijker gezocht worden. Full-text zet alles om naar tekst en maakt een index aan per entiteit. Zo worden alle data als tekst opgeslagen en kan worden bijgehouden hoe vaak woorden voorkomen. Dit heeft ook een aantal problemen tot gevolg.

Het eerste probleem is de structural mismatch. Doordat alles wordt omgezet naar tekst gaat er ook een hoop informatie verloren. Types van de verschillende kolommen zijn niet meer bekend, alles is immers naar tekst omgezet, en verwijzingen tussen verschillende documenten zijn verloren gegaan. De in de database vastgelegde structuur is grotendeels verloren gegaan in de index.

De zoekdata (indexen) en de echte data (database) worden los van elkaar opgeslagen, en hebben geen weet van elkaar. Om de indexen up-to-date te houden moet extra werk verzet worden. Als de werkelijke data in de database veranderen, worden de verschillende indexen niet automatisch bijgewerkt. Dit kan tot verkeerde zoekresultaten leiden. Dit noemen we de synchronization mismatch. Er moet extra code worden geschreven om de indexen te updaten als de werkelijke data veranderen. Het liefst natuurlijk op een manier die de performance van de applicatie niet beïnvloedt. Hierbij moet een afweging worden gemaakt tussen de achteruitgang van de performance tijdens het indexeren of de accuratesse van de zoekresultaten.

Ten slotte is er nog de retrieval mismatch. Het zoekresultaat van een full-text index omzetten naar een Java-object is niet triviaal. Om het resultaat van een zoekquery te kunnen gebruiken als persistent object, een object dat door Hibernate sessie of EntityManager wordt gemanaged, is nog een stapje verder. Veel full-text search frameworks geven je een Java-object, maar dat 'mapt' nog niet naar je domain-model. Er moet nog extra code worden geschreven om het zoekresultaat ook te gaan gebruiken.

Hibernate Search en Lucene

Eind september vorig jaar kondigde Emmanuel Bernard, de projectlead van Hibernate Search, op zijn blog aan dat Hibernate Search final was.

Hibernate Search is een laag die tussen Lucene, de bekendste Java full-text search-engine en Hibernate zit. Ontwikkelaars die al bekend zijn met Lucene en Hibernate of JPA zullen hierdoor snel aan de slag kunnen. Om een beter begrip te krijgen van wat je met Hibernate Search kunt is het nuttig om even kort in Lucene te duiken; of om je kennis op te frissen. Om met Lucene data te kunnen doorzoeken, moet deze eerst geïndexeerd worden. De index is niets meer dan een aantal bestanden op de harde schijf waarmee Lucene sneller de data kan doorzoeken. Een Lucene-index bestaat uit document-objecten die fields bevatten. Een field is niets meer dan een key-value pair zoals we die van map-objecten kennen. Een zoekvraag heet een query en heeft vrijwel dezelfde syntax als Google-zoekopdrachten Anders dan bij SQL zijn queries zo simpel dat eindgebruikers en ontwikkelaars dezelfde queries kunnen gebruiken.

Met Lucene kunnen Google-like queries geschreven worden. Geen ingewikkelde queries, in het simpelste geval een paar woorden en in complexere gevallen op veldnamen en gebruikmakend van + en – om aan te geven dat iets juist wel of niet gevonden mag worden; bijvoorbeeld site:sun.com javadoc -string +buffer. Stel dat we een geïndexeerde dataset hebben met een aantal personen erin. De beschikbare velden zijn firstName, lastName, birthYear en gender. Bij een Lucene-query geef je een default zoekveld op, dat is in ons geval lastName.

Query	**Resultaat**
jansen pietersen	Alle personen met als achternaam Jansen **of** Pietersen.
firstName:robin AND gender:male	Alle mannen met Robin als voornaam.
+gender:female -firstName:kobie	Alle vrouwen die niet Kobie als voornaam hebben
b*k	Iedereen waarvan de achternaam begint met een b en eindigt met k (bv. Beek, Broek)
birthYear:1952 gender:female	Alle vrouwen geboren in 1952
birthYear:[1980 TO 2000]	Iedereen die geboren is van 1980 **tot en met** 2000 (inclusief)
birthYear:{1980 TO 2000}	Iedereen geboren **tussen** 1980 en 2000, dus zonder 1980 en 2000 (exclusief)

Met Lucene-queries kun je Booleaanse logica gebruiken en/of de +/- notatie. Daarnaast kent Lucene nog proximity-search, stemming, fuzzy search en boosting van zoektermen. Voor meer informatie over Lucene-queries kun je kijken op http://lucene.apache.org/java/docs/queryparsersyntax.html. Belangrijk om in het achterhoofd te houden is dat queries voor Lucene altijd gebaseerd zijn op strings. Als je op getallen wilt zoeken, zul je deze met voorloopnullen moeten opslaan.

Entiteiten mappen

Hibernate Search heeft een aantal annotaties beschikbaar die aan de domeinobjecten kunnen worden toegevoegd. Deze annotaties geven aan hoe Lucene een object moet indexeren. Als voorbeeld gebruiken we een Person class. Deze class is voorzien van de nodige JPA-annotaties om de verschillende properties naar de database te mappen. Om deze class ook te laten indexeren door Lucene moeten er ook een aantal Hibernate Search-annotaties worden toegevoegd. Dit zal veelal op dezelfde plekken zijn als waar de JPA-annotaties zijn geplaatst.

Op class-niveau moet er een @Indexed-annotatie worden toegevoegd om aan te geven dat er voor deze class een index moet worden aangemaakt. Eventueel kan de annotatie een argument-index meekrijgen om de index een specifieke naam te geven. Default is dit de fully qualified name van de class. De index wordt normaal gesproken op het filesysteem weggeschreven. Er zal een directory met de fully qualified name worden aangemaakt in de index directory van Lucene.

Net als bij een entity moet worden aangegeven wat de id is van de index. Dit gebeurt met de @DocumentID-annotatie. Dit moet de identifier-property van het te indexeren object zijn. In vrijwel alle gevallen zal dit dezelfde property zijn als de primary key (@Id) van de entity.

```
 @Entity
@Indexed
public class Person {

    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Basic
    @Field
    private String firstName;

    @Basic
    @Field
    private String lastName;

    @Temporal(TemporalType.DATE)
    @Field(index = Index.UN_TOKENIZED)
    @DateBridge(resolution = Resolution.DAY)
    private Date birthdate;

    @Enumerated(EnumType.STRING)
    @Field
    private Gender gender;

    @ManyToOne
    @IndexedEmbedded(depth = 1)
    private Address address;
```

Met het store-argument kun je aangeven of de waarde in het Lucene-document moet worden opgeslagen

Per property kan vervolgens worden aangegeven of deze in de index moet worden meegenomen, en hoe. Als een property in de index wordt opgenomen, kunnen we op dit veld zoeken. Door een @Field-annotatie boven een property te zetten wordt het veld door Lucene geïndexeerd. Hoe Lucene het veld precies indexeert, kan je met een aantal argumenten meegeven aan de annotatie. In de meeste gevallen zullen de defaults echter voldoende zijn. Met het index-argument kan worden aangegeven of het veld moet worden opgesplitst in aparte woorden (tokenized) of dat het een lange string moet blijven (un_tokenized). De naam van de property die uiteindelijk in de zoekquery gebruikt kan worden, is default de naam van de property zelf. Met het store-argument kun je aangeven of de waarde in het Lucene-document moet worden opgeslagen. Default is dit no, wat betekent dat je alleen kunt zoeken op dit veld en niet de waarde van het veld kunt opvragen. In eerste instantie klinkt het vreemd dat je de waarde van het veld niet meer kunt zien, maar databases werken op dezelfde manier. Hier kun je via de index razendsnel de positie van het gezochte record ophalen en pas daarna wordt het uiteindelijke record opgehaald.

Met de optie 'yes' kun je de oorspronkelijke waarde van het veld nog achterhalen en zal er ook een index-versie van het veld opgeslagen worden om snel te kunnen zoeken. Een andere mogelijkheid is compressed. Hiermee wordt ruimte bespaard, vergeleken met de yes-strategie, maar wordt meer van de cpu gevraagd tijdens het indexeren en zoeken. Dit kan gebruikt worden voor grote documenten of binaire bestanden.

Met de @Fields-annotatie kan een property meerdere keren geïndexeerd worden binnen dezelfde Lucene-index met steeds een iets andere strategie. Dit is bijvoorbeeld nodig als het zoekresultaat gesorteerd moet worden. Bij een tokenized strategie worden de verschillende woorden los in de index opgeslagen, zodat bij het sorteren dus een un_tokenized strategie zou moeten worden gebruikt. Om dit beide mogelijk te maken kan er binnen een @Fields-annotatie twee verschillende @Field-annotaties worden gezet met elk een ander name- en index-argument.

```
@Entity
@Indexed
public class Address {
     @Id
     @GeneratedValue
     @DocumentId
     private Long id;

     @Field(index=Index.TOKENIZED)
     private String street;

     @Field
     private Long houseNumber;

     @Field(index=Index.TOKENIZED)
     private String city;

     @ContainedIn
     @OneToMany
     private Set<Person> persons;

     ...
}
```

Bij een referentie naar een ander object kan de @IndexEmbedded-annotatie worden gebruikt. Door het toevoegen van deze annotatie worden de properties die zijn voorzien van een @Field-annotatie, ook meegenomen in de index. Om op deze properties te querien kan de standaard objectnavigatie (address.city) gebruikt worden. Eventueel kan bij de @IndexdEmbedded-annotatie een andere prefix worden opgegeven. In het geval van het address-object in person zou een @IndexEmbedded(prefix = "address_") annotatie er voor zorgen dat op het veld city kan worden gezocht. Je moet hiervoor dan wel address_city in de query gebruiken. Het depth-argument geeft aan hoe diep Hibernate Search gaat met het toevoegen van properties aan de index. Standaard is dit oneindig. Bij een cyclische verwijzing treedt een exceptie op. Het object dat met een @IndexedEmbedded-annotatie wordt opgenomen in de index kan zelf ook een index definiëren door een @Index boven de class te zetten, maar dit is niet verplicht. Bij het address-voorbeeld is dit wel gedaan. Lucene zal dan ook twee indexen aanmaken, een voor person en een voor address.

Het embedded index-object, Address in het voorbeeld moet een bidirectionele verwijzing hebben. Deze moet worden voorzien van een

@ContainedIn-annotatie. Alleen op deze manier weet Hibernate Search dat als het Address-object wijzigt, ook de index van de Person-instanties moet worden bijgewerkt. De @IndexEmbedded wordt ook ondersteund op collecties, maar dit is nog niet volledig geïmplementeerd. Hibernate Search zal niet altijd de index updaten van het object naar welke wordt verwezen. Dit kan voorlopig worden opgelost door zelf het object opnieuw te indexeren.

Indexeren

Zodra geannoteerde entiteiten worden gepersisteerd, gewijzigd of verwijderd via de session- of entity manager, zorgt Hibernate Search er automatisch voor dat ook de index bijgewerkt wordt. Hibernate Search is nog net niet slim genoeg om te zien dat je een initiële index op een bestaande database wilt maken. Om de initiële set data te laten indexeren, zal elk record de FullTextEntityManager moeten passeren. In het volgende voorbeeld wordt één Person-object in de index opgeslagen door middel van de index-methode op de FullTextEntityManager.

```
EntityManager em = emFactory.createEntityManager();
Person person = em.find(Person.class, Long.
valueOf(1L));    FullTextEntityManager ftem = Search.
createFullTextEntityManager(em);              ftem.
index(person);
```

De FullTextEntityManager of de FullTextSession hebben ook de mogelijkheid om handmatig een object of alle indexen van een class te verwijderen: ftem.purgeAll(Person.class).

```
FullTextEntityManager ftem = Search.createFullTextEn
tityManager(em);
ftem.purgeAll(Person.class);
ftem.purge(Person.class, person.getId());
```

De belangrijkste annotaties zijn nu aan bod geweest en de indexen voor de persoon- en adresobjecten worden nu automatisch aangemaakt. Lucene biedt ook nog andere functionaliteit die ook in Hibernate Search beschikbaar is. Hieronder zullen we nog wat andere annotaties bespreken die Hibernate Search ook biedt.

Boost

Bij het zoeken in Lucene zullen de resultaten op score gesorteerd worden. Deze score wordt via een lastig algoritme berekend. Stel dat je de query "fiets bel" uitvoert, dan zullen documenten waarin beide woorden voorkomen hoger in het resultaat terechtkomen dan de documenten waar maar één van de twee woorden in voorkomt. Om het berekenen van de score te beïnvloeden kun je boosting gebruiken. Met een boost-factor kun je aangeven dat een bepaald veld of bepaalde klasse belangrijker (of juist minder belangrijk) is dan een andere bij een zoekactie. Het is ook mogelijk om al tijdens het indexeren aan te geven hoe belangrijk een field is. Zo'n boost-factor kun je op een class of field zetten met @Boost(waarde). Met een waarde van 0.5 geef je aan dat de rest van de velden twee keer zo belangrijk is dan het veld dat voorzien is van de annotatie.

Analyzer

In dit artikel zul je een aantal keer het object Analyzer (of een variant daarop) tegengekomen zijn. Een analyzer zorgt ervoor dat de data beter doorzoekbaar worden door het opsplitsen van strings in woorden, het herschrijven van hoofdletters naar kleine letters, het verwijderen van leestekens en eventuele custom-stappen. Er zijn ook geavanceerdere analyzers die bijvoorbeeld accenten van letters verwijderen zodat één en hetzelfde woord ontstaat. Ook het aanpassen van de analyzer is een kleine moeite. Als niets is opgegeven wordt de StandardAnalyzer gebruikt. Deze Analyzer maakt van alle letters kleine letters, laat alle leestekens weg, splitst woorden op spaties en laat bekende Engelse woorden weg (de zogenaamde stop-words). Het is mogelijk de @Analyzer-annotatie te gebruiken. Dan kun je per veld of klasse instellen welke analyzer je wilt gebruiken. Houdt er bij het aanpassen van de analyzer rekening mee dat je ook voor het zoeken dezelfde analyzer gebruikt. Als je tijdens het indexeren accenten van letters weglaat, en je doet dit niet tijdens het zoeken, dan zal dit een fout zoekresultaat opleveren. Over het algemeen zul je maar één analyzer gebruiken en het gemakkelijkste is het om deze als property (hibernate.search.analyzer) in te stellen. Via de site van Lucene is een aantal Analyzers beschikbaar, vooral voor verschillende talen (bijvoorbeeld Chinees, Frans, Duits, Grieks en Russisch). Er is ook een DutchAnalyzer beschikbaar. Deze slaat veelvoorkomende Nederlandse woorden over tijdens het indexeren.

FieldBridge

Omdat Lucene alles naar strings indexeert, loop je bij getallen al snel tegen problemen aan. Als er op de reeks 10 t/m 20 gezocht wordt, zal het getal 100 ook gevonden worden. Een oplossing voor dit probleem is het gebruik van voorloopnullen. In Lucene moest je hier tijdens het indexeren al rekening mee houden. Met Hibernate Search kun je met de @FieldBridge-annotatie aangeven dat er eerst iets met de veldwaarde moet gebeuren. De @FieldBridge zet je op de getter of op de property zelf, net als de @Field-annotatie.

```
@Column
@Field(store=Store.YES)
@FieldBridge(imp = ZeroPadder.class)
public Long getHouseNumber() {
    return houseNumber;
}
public class ZeroPadder implements StringBridge {
    public String objectToString(Object o) {
        Long l = (Long) o;
        return String.format("%05d", l);
    }
}
```

De klasse bij de FieldBridge is een implementatie van StringBridge die de methode objectToString(Object o) bevat. Deze methode zet het input-object om naar een string zodat Lucene het verschil tussen 100 en 10 gaat begrijpen. Helaas worden de queries (nog) niet vanzelf aangepast. Hiervoor zul je nog een eigen QueryParser moeten maken. Als we nu de query housenumber:[10 TO 20] uitvoeren, zien we echt een vreemd resultaat. Alleen de huisnummers van 10000 tot en met 20000 worden gevonden. De QueryParser van Lucene moet ook nog aangepast worden op het gebruik van voorloopnullen. Het aanpassen van een QueryParser is relatief simpel. Extend een bestaande QueryParser en override de methode voor de RangeQuery (de queries waar TO en {} of [] gebruikt worden). Gebruik nu deze parser bij het uitvoeren van de queries en je zult merken dat de queries herschreven zijn en nu goed werken. Het zoeken op getallen is een praktijkvoorbeeld waar je snel mee te maken zult krijgen.

```
public class ZeroQueryParser extends QueryParser {
    public ZeroQueryParser(String s, Analyzer ana-
lyzer) {
        super(s, analyzer);
    }

     protected Query getRangeQueryf(String field,
String part1, String part2,
        boolean inclusive) throws ParseException {

        if (field.equals("housenumber")) {
            String empty = "00000";
    String result1 = empty.substring(part1.length()) +
part1;
    String result2 = empty.substring(part2.length()) +
part2;
      return new RangeQuery(new Term(field, result1),
new Term(field, result2), inclusive);
        }

     return super.getRangeQuery(field, part1, part2,
inclusive);
    }
```

Een andere toepassing van een FieldBridge is wanneer een pdf-file opgenomen is in een object. Ook de pdf zou je willen doorzoeken met Hibernate Search. Met een FieldBridge kun je dan aangeven hoe een pdf-bestand moet worden geïndexeerd door het met een pdf-extractor om te zetten naar een string. Zo kan het document toch doorzocht worden. Data en tijd zijn ook zaken die je in de praktijk veel tegenkomt. Hiervoor kun je de @DateBridge gebruiken. Hibernate Search slaat data op in de yyyyMMddHHmmssSSS-notatie. Milliseconden kunnen handig zijn, maar als je een geboortedatum van iemand opslaat heb je alleen de dag nodig. De @DateBridge heeft een attribuut-resolution waarin je de nauwkeurigste waarde zet die je wilt opslaan. In het geval van een geboortedatum zou je @DateBridge(resolution =Resolution.DAY) gebruiken. Met Resolution. DAY worden alleen dag, maand en jaar in de index gebruikt.

Querying

Voor het querien van de Lucene-index biedt Hibernate Search een wrapper die dit eenvoudiger maakt. Dit kan op basis van een Hibernate-session of een JPA-entityManager. Dit gebeurt vrijwel op dezelfde manier, met als enig verschil dat de package-naam van de Hibernate Search-objecten `org.hibernate.search` is en die van de JPA-laag `org.hibernate.search.jpa`. De meeste in dit artikel gebruikte voorbeelden zullen de JPA-aanpak gebruiken, maar deze zijn net zo eenvoudig met een Hibernate-session te schrijven.

```
FullTextEntityManager ftem = Search.createFullTextEn
tityManager(em);
List<?> list = ftem.createFullTextQuery(luceneQuery)
.getResultList();
Of
FullTextSession     ftSession     =     Search.
createFullTextSession(session);
List<?> list = ftSession.createFullTextQuery(luceneQ
uery).getResultList();
```

Pagination

Een andere belangrijke functie die de FullTextQuery ondersteunt is de toepassing van paginering. Op deze manier is het eenvoudig door een resultSet te scrollen. Door het zetten van de maxResults en de FirstResult zal alleen een beperkte set van de data worden opgehaald. Op de FullTextQuery is ook de methode getResultSize() aanwezig om eenvoudig het aantal resultaten te bereken zonder alle objecten aan te maken. Hierdoor kan eenvoudig worden getoond hoeveel het totaal aantal zoekresultaten is en hoeveel pagina's met resultaten er nog beschikbaar zijn.

```
     FullTextQuery     ftQuery     =     ftem.
createFullTextQuery(query, Person.class);
ftQuery.setMaxResults(10);
ftQuery.setFirstResult(0);
```

Sorting

Het aangeven van de sortering die op het resultaat van een query moet zitten, is eenvoudig te

doen. Het kan op een veld, maar ook meerdere velden die in de index voorkomen. Op het FullTextQuery-object kan een sort-property worden gezet. Een sort-object kan op een of meerdere velden uit de query opgebouwd zijn. Met een Boolean wordt aangegeven of het SortField oplopend of aflopend is.

```
FullTextQuery ftQuery = ftem.
createFullTextQuery(query, Person.class);
Sort sort = new Sort(new SortField("firstName",
false));
ftQuery.setSort(sort);
```

Projection

Projection is een manier om niet het hele object op te halen dat de zoekquery oplevert, maar enkel een aantal properties van het resultaat. Voorwaarde is wel dat deze properties met een @Field(store = Store.YES) zijn opgenomen. Als deze niet worden opgeslagen - wat default is - zal toch eerst het hele object moeten worden opgehaald. Projection is dus vooral handig bij grote datasets waarvan niet het hele object nodig is, maar bijvoorbeeld alleen de voornaam en geboortedatum.

Configuratie

Om Hibernate Search te kunnen gebruiken heb je de 'jars' van Lucene, Hibernate Search, Hibernate en Hibernate Annotations nodig. Als Maven 2 wordt gebruikt om de projectafhankelijkheden te organiseren is het voldoende om org.hibernate.hibernate-search en org.hibernate.hibernate-annotations toe te voegen.

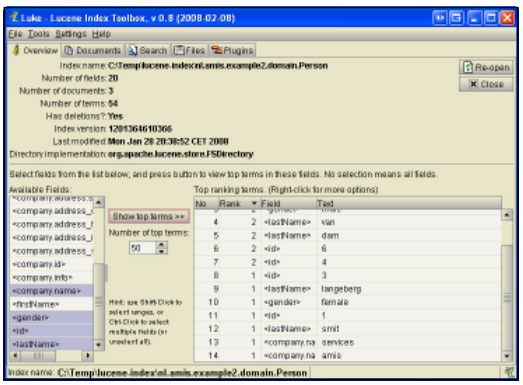
```
FullTextQuery ftQuery = ftem.createFullTextQuery(luc
eneQuery, Perons.class);
ftQuery.setProjection("firstName","birthDate","addre
ss.city");
List results = ftQuery.getResultList();
Object[] firstResult = (Object[]) results.get(0);
String firstName = firstResult[0];
Date birthDate = firstResult[1];
String city = firstResult[2];
```

Het configureren van de verschillende indexen kan op dit moment alleen via annotaties en niet in de Hibernate configuratie-file. Misschien dat dit in de toekomst nog wel wordt toegevoegd. Tijdens afgelopen JavaPolis zei Emmanuel Bernard dat er een persoon was opgestaan die dit graag wilde doen. Andere configuratie vindt plaats in de hibernate.properties of persistence.xml. De property *hibernate.search.default.directory_provider* geeft aan wat voor configuratie wordt gebruikt. De property initialiseert ook de Lucene-directory. In het geval van een directory-provider kan ook de locatie worden aangegeven. Een andere mogelijkheid is om ervoor te kiezen de Lucene-indexen in het geheugen op te slaan in plaats van op de harddisk. Dit is vooral een handige optie met testen.

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search</artifactId>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
  </dependency>
</dependencies>
```

Tools

Met een SQL-database heb je een command-line of GUI om de data snel te kunnen controleren. Om Lucene-indexen te kunnen bekijken is de tool Luke beschikbaar. Luke is een Swing-client waarmee je door je Lucene-documenten kunt bladeren of queries op jouw index kunt uitvoeren. Een andere handige optie van Luke is het snel kunnen wisselen van analyzer. Lucene herschrijft queries voor intern gebruik naar de +/- notatie. Dit is te bekijken met explain-structure; een soort explain-plan van de query.
Een andere tool is LIMO (Lucene Index Monitor). Dit is een webapplicatie waarmee je ook door de index kunt bladeren en simpele queries kunt uitvoeren. Deze tool is vooral handig om te controleren of documenten geïndexeerd zijn. LIMO reageert een stuk sneller op zoekopdrachten en het bladeren door de index dan Luke, maar is wel wat beperkter in de zoekopdrachten. De laatste versie van LIMO is van januari 2007 en gebruikt een versie van Lucene die niet geschikt is voor de indexen die met Hibernate Search gemaakt zijn. Dit is simpel op te lossen door lucene-core-2.0.0.jar (in WEB-INF\lib) te overschrijven door de nieuwste versie. Ook kan het soms nodig zijn de index te optimaliseren als er problemen optreden.



```
hibernate.search.default.directory_provider=org.
hibernate.search.store.FSDirectoryProvider
hibernate.search.default.indexBase=/usr/lucene/
indexes

hibernate.search.Rules.directory_provider=org.hiber-
nate.search.store.RAMDirectoryProvider
```

Conclusie

Lucene is een product dat je redelijk snel onder de knie hebt, maar je moet zelf veel van de 'plumbing' doen. Ook zul je veel acties vaak moeten herhalen. Het is gemakkelijk om te vergeten een index te updaten als een object verandert of wordt toegevoegd. Hibernate Search lost een aantal belangrijke problemen op die anders de nodige code vereisen. Je hoeft je niet meer druk te maken om het synchronisatieprobleem. Wordt een entity door Hibernate of JPA toegevoegd, veranderd of verwijderd, dan zal Hibernate Search er ook voor zorgen dat de eventuele indexen die op de class zitten ook worden geüpdate. Lucene-documenten behoren tot het verleden. Je krijgt na een query gelijk de persistent objecten met hun eventuele relaties tot je beschikking. Om bestaande code aan te passen, zodat ook die gebruik kan maken van Lucene, zijn slechts minimale wijzigingen nodig. Mits Hibernate of JPA al worden gebruikt. Het resultaat van een 'normale' query en een Lucene-query is hierdoor in de code transparant geworden.

Tegenwoordig wordt er bij open source-producten steeds meer aandacht aan de documentatie besteed. De documentatie van Hibernate Search is erg goed op orde. Problemen waar je in de praktijk tegenaan loopt zijn duidelijk beschreven. Een vraag op het Hibernate-forum werd binnen een dag beantwoord en de lead developer is ook erg actief op het forum. Dit is een mooie toevoeging aan Hibernate en misschien in de toekomst ook wel voor JPA. Aangezien Emmanuel lid is van de expertgroep is het niet ondenkbaar dat dergelijke functionaliteit in de Java-persistent API wordt opgenomen. «

Bronnen

Jboss blog: http://in.relation.to/
Hibernate Search: http://search.hibernate.org/
Lucene: http://lucene.apache.org
Luke: http://www.getopt.org/luke/
Limo: http://limo.sourceforge.net/
Lucene query syntax: http://lucene.apache.org/java/2_3_0/queryparsersyntax.html
Lucene scoring: http://lucene.apache.org/java/2_3_0/scoring.html
Zoeken op numerieke velden in Lucene: http://wiki.apache.org/lucene-java/SearchNumericalFields