

Logo 1

Logo2

Universität Bayreuth
Fakultät für Informatik

Bachelorarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades
Bachelor / Master of Science

Thema: Integration of JPA-conform ORM-Implementations in Hibernate Search

Autor: Martin Braun <martinbraun123@aol.com>
Matrikel-Nr. 1249080

Version vom: July 23, 2015

1. Betreuer: Dr. Bernhard Volz
2. Betreuer: Prof. Dr. Bernhard Westfechtel

Zusammenfassung

Abstract

Contents

1	Preface	5
2	Overview	7
2.1	An overview of different database paradigms	7
2.1.1	relational databases	7
2.1.2	NoSQL databases	7
2.2	Object Relational Mappers	7
2.2.1	JPA	7
2.3	Fulltext search	8
2.3.1	Lucene	9
2.3.2	Solr	10
2.3.3	ElasticSearch	10
2.3.4	Hibernate Search	10
2.4	aims of this thesis	10
3	Showcase of basic Hibernate Search	10
4	Ausblick	10
5	Fazit	10
	Literaturverzeichnis	13
	Anhang	14
	Eidesstattliche Erklärung	14

1 Preface

In the software world, or more specific, the Java enterprise world, developers tend to abstract access to data in a way that components are interchangeable. A perfect example for such an abstraction is the usage of Object Relational Mappers (ORM). The database specifics are mostly irrelevant to the average developer and the need for native SQL is brought down to a minimum. This makes the switch to a different relational database system (RDBMS) easier in the later stages of a product's life cycle.

The Java Persistence API (JPA) went even further by standardising ORMs. First conceived in 2006 [?], it is now the de-facto standard for Object Relational Mappers in Java. The developer doesn't need to know which specific ORM is used in the application, as all the database queries are written against a standardized query API and therefore portable. This means that not only the database is interchangeable, but even the specific ORM, it is accessed by, is as well.

However, this does not mean that all JPA implementations ship with the same features. While all of them are JPA compliant (apart from minor bugs), some ship with additional modules to enhance their capabilities. A perfect example for this is the Hibernate Search API aimed at Hibernate ORM users: Nowadays, even small applications like online shops need enhanced search capabilities to let the user find more results for a given input.

This is not something a regular RDBMS excels at and Hibernate Search comes into use: It works atop the Hibernate ORM/JPA system and enables the developer to index the domain model for searching. It's not only a mapper from JPA entities to a search index, but also keeps the index up-to-date if something in the database changes.

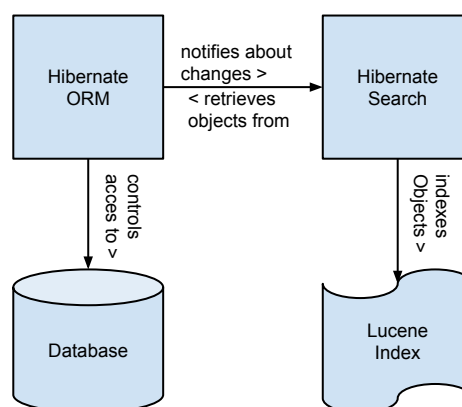


Figure 1: Hibernate Search with Hibernate ORM

Hibernate Search, which is based on the powerful Lucene search toolbox, is a separate project in the Hibernate family and is using a lot of JPA interfaces in its codebase and aims to provide a JPA "feeling" in its API. However, this does not mean that it is compatible with other JPA providers than Hibernate ORM (apart from Hibernate OGM, the NoSQL JPA mapper of the family).

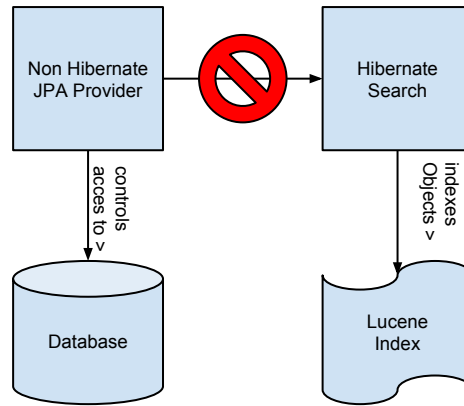


Figure 2: Hibernate Search's incompatibility with other JPA implementations

While using Hibernate Search obviously is beneficial for Hibernate ORM applications, not all developers can bind themselves to a specific JPA implementation in their application. For some, the ability to change the it is of strategic important, for others it is just sheer preference to use a different JPA implementation.

Currently these developers have to resort to using different full text search systems like native Lucene, ElasticSearch or Solr. While this is always a viable option, for some applications Hibernate Search would be a much better suit because of it's design with a entity structure in mind and the automatic index updating feature, if it just were compatible with generic JPA.

When investigating Hibernate Search's project structure [?], we see that the only module apart from some server-integration modules that depends on any ORM logic is "hibernate-search-orm". The modules that contain the indexing engine, the replication logic, alternative backends, etc. are completely independent from any ORM logic. This means, that we can reuse most of the codebase for a generic version of Hibernate Search.

In this thesis we will show how such a generic version can be built. We will look at how Hibernate Search's engine can be reused. Then, we will write a standalone version of this engine and finally integrate it with generic JPA.

2 Overview

2.1 An overview of different database paradigms

das hier anders

When it comes to persisting data in applications, nowadays there exist a lot of different paradigms that one has to choose from. Following is a short explanation for the two currently most used ones.

2.1.1 relational databases

2.1.2 NoSQL databases

2.2 Object Relational Mappers

While the NoSQL approach is undeniably rising in popularity, the demand for relational solutions is still unmatched, as it has been used and has proven itself in practice for many years now.

Nowadays, many popular languages like Java, C#, etc. are object-oriented. While SQL solutions for querying relational databases exist for these languages, the user either has to work with the rowsets manually or convert them into custom data access objects. Both approaches include a lot of manual work.

This is where Object Relational Mappers (ORM) come into use. They map tables to entity-classes and enable users to write queries against these classes instead of tables. This is especially useful if used in big software products as not all programmers have to know the exact details of the underlying database. The database system could even be completely replaced for another (provided the ORM supports the specific RDBMS), with the business logic not changing a bit.

2.2.1 JPA

The first version of the JPA standard was released in May 2006. From then on it rose to probably the most commonly used persistence API for Java. While mostly known for standardizing relational database mappers (ORM), it supports other concepts like NoSQL or XML storage as well. However, when talking about JPA in this thesis we will be focusing on the relational aspects of it. Currently, the newest version of this standard is 2.1.¹

Some popular relational implementations are:

¹Wikipedia on Java Persistence API, see [1]

- Hibernate ORM (JBoss)
- EclipseLink (Eclipse foundation)
- OpenJPA (Apache foundation)

vll Beispiel für eine einfache Beziehung, ER-Modell vs. gemappte Klasse

Using the standardized JPA API over any native ORM API has one really interesting benefit: The specific JPA implementation can be swapped out. This is particularly important if you are working in a Java EE environment. Java EE itself is a specification for platforms, mostly Web-servers (JPA is part of the Java EE spec).² Many Java EE Web-servers ship with a bundled JPA implementation that they are optimized for. This means that if a user switches servers, he/she is also likely to swap out the JPA implementor. If everything in the application is written in a JPA compliant way, the user will then generally not run into many problems related to this switch.

2.3 Fulltext search

Conventional relational databases are good at retrieving and querying structured data. But if one wants to build a search engine atop a domain model, most RDBMS will only support the SQL-LIKE operator:

```
1 SELECT book.id FROM book WHERE book.name LIKE %name%;
```

While this might be enough for some applications, this wildcard query doesn't support features a good search engine would need, for example:

- fuzzy queries (variations of the original string will get matched, too)
- phrase queries (search for a specified phrase)
- regular expression queries (matches are determined by a regular expression)

There may exist some RDBMS that support similar query-types, but in the context of using a ORM we would then lose the ability to switch databases since we require specific features not every RDBMS supports.

Fulltext search engines can be used to complement databases in this regard. They are not intended to be replacing the database, but to add additional functionality by indexing the data that is to be searched in a more sophisticated way. We will now take a look at some of the most popular available options for Java developers while focusing on their usage, features, the pros and cons of using them, and compatibility with the JPA standard.

²Wikipedia on Java EE, see [2]

2.3.1 Lucene

mention current version for each of these?

Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.³

Lucene serves as the basis for most fulltext search engines written in Java. It has many different utilities and modules aimed at search engine developers. However, it can be used on its own as well.

Index structure Lucene uses an **inverted index** to store data. This means that instead of storing texts mapped to the words contained in them, it works the other way around. All different words (or terms) are mapped to the texts they occur in.⁴ Also, before anything can be searched using Lucene, it has to be added to the the index first.

Concepts Lucene has its own set of concepts that need to be discussed first before we can take a look at it's usage. Following is the explanation of the most important ones.

Documents Documents are the data-structure Lucene stores and retrieves from the index. A index can contain zero or more Documents. Documents are added to the index with an IndexWriter and retrieved via an IndexReader/IndexSearcher.

Fields A Document consists of at least one field. Fields are basically tuples of key and value. They can be stored (can be retrieved from the index) and/or indexed (can be searched on).

Analyzers Before documents get indexed, their fields are analyzed first. Analysis is the process of modifying the input in a manner such that it can be searched upon (stemming, tokenization, ...). In Lucene this is done by special classes called Analyzers.

Usage - Indexing We will now take a look at how data is indexed in Lucene. In the following example we consider the data to be already present in form of a List of objects of the class 'Text' and concentrate on the Lucene usage itself.

Beispiel für Lucene usage hier,

³official Lucene website, see [3]

⁴Lucene basic concepts, see [4]

Usage - Searching Searching

Beispiel für Lucene usage hier,

Features Lucene is probably the most complete toolbox to build a search-engine from.

Pros and Cons

Compatibility with JPA By design, Lucene out of the box is not very compatible with the JPA standard. For one, the flat document structure forces the user to de-normalize the entity model before indexing. Secondly, since every search-relevant change in the database should be reflected in the index, it must be kept up to date. When using Lucene, this has to be done completely manually as it natively doesn't have any integration with databases.

2.3.2 Solr

2.3.3 ElasticSearch

2.3.4 Hibernate Search

some kind of conclusion with a table of features. -> Hibernate Search, aber mit dem Problem von Kompatibilität mit Non Hibernate ORM, mention Compass?

2.4 aims of this thesis

3 Showcase of basic Hibernate Search

4 Ausblick

5 Fazit

Abbildung ?? [S.??]

Überschrift 1	Überschrift 2
Info 1	Info 2
Info 3	Info 4

```

1 <dataConfig>
2   <dataSource type="JdbcDataSource"
3       driver="com.mysql.jdbc.Driver"
4       url="jdbc:mysql://localhost/bms_db"
5       user="root"
6       password="" />

```

```

7 <document>
8   <entity name="id"
9     query="select id, htmlBody, sentDate, sentFrom, subject, textBody
10    from mail">
11   <field column="id" name="id"/>
12   <field column="htmlBody" name="text"/>
13   <field column="sentDate" name="sentDate"/>
14   <field column="sentFrom" name="sentFrom"/>
15   <field column="subject" name="subject"/>
16   <field column="textBody" name="text"/>
17   </entity>
18 </document>
19 </dataConfig>

```

Listing 1: Die Datei data-config.xml dient als Beispiel für XML Quellcode

```

1  /* generate TagCloud */
2  Cloud cloud = new Cloud();
3  cloud.setMaxWeight(_maxSizeOfText);
4  cloud.setMinWeight(_minSizeOfText);
5  cloud.setTagCase(Case.LOWER);
6
7  /* evaluate context and find additional stopwords */
8  String query = getContextQuery(_context);
9  List<String> contextStoplist = new ArrayList<String>();
10 contextStoplist = getStopwordsFromDB(query);
11
12 /* append context stoplist */
13 while(contextStoplist != null && !contextStoplist.isEmpty())
14   _stoplist.add(contextStoplist.remove(0));
15
16 /* add cloud filters */
17 if (_stoplist != null) {
18   DictionaryFilter df = new DictionaryFilter(_stoplist);
19   cloud.addInputFilter(df);
20 }
21 /* remove empty tags */
22 NonNullFilter<Tag> nnf = new NonNullFilter<Tag>();
23 cloud.addInputFilter(nnf);
24
25 /* set minimum tag length */
26 MinLengthFilter mlf = new MinLengthFilter(_minTagLength);
27 cloud.addInputFilter(mlf);
28
29 /* add taglist to tagcloud */
30 cloud.addText(_taglist);
31
32 /* set number of shown tags */
33 cloud.setMaxTagsToDisplay(_tagsToDisplay);

```

Listing 2: Das Listing zeigt Java Quellcode

Die Zuordnung aller möglichen Werte, welche eine Zufallsvariable annehmen kann nennt man *Verteilungsfunktion* von X .

Die Funktion $F: \mathbb{R} \rightarrow [0,1]$ mit $F(t) = P(X \leq t)$ heißt Verteilungsfunktion von X .⁵

Für eine stetige Zufallsvariable $X: \Omega \rightarrow \mathbb{R}$ heißt eine integrierbare, nicht-negative reelle Funktion $w: \mathbb{R} \rightarrow \mathbb{R}$ mit $F(x) = P(X \leq x) = \int_{-\infty}^x w(t)dt$ die *Dichte* oder *Wahrscheinlichkeitsdichte* der Zufallsvariablen X .⁶

⁵Konen, vgl. [?] [S.55]

⁶Konen, vgl. [?] [S.56]

Literaturverzeichnis

Anhang

Literaturverzeichnis

- [1] Wikipedia https://en.wikipedia.org/wiki/Java_Persistence_API,
07/16/2015
- [2] Java Platform, Enterprise Edition Wikipedia https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition, 07/16/2015
- [3] Lucene Website <https://lucene.apache.org/core/>, 07/16/2015
- [4] Lucene Tutorial <http://www.lucenetutorial.com/basic-concepts.html>,
07/20/2015

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur <-Arbeit>

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift :

Ort, Datum :

