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 29, 2015
1. Betreuer:	Dr. Bernhard Volz
2. Betreuer:	Prof. Dr. Bernhard Westfechtel

Zusammenfassung

Abstract

Contents

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 [1], 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 come 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.[2] [3]

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.

[1] Wikipedia on Java Persistence API, see [1]
[2] Hibernate ORM project homepage, see [8]
[3] Hibernate Search project homepage, see [2]

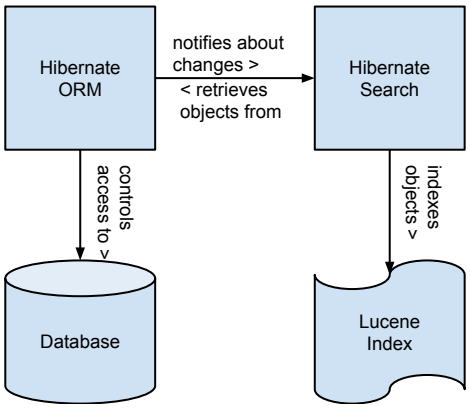

Figure 1: Hibernate Search with Hibernate ORM

Hibernate Search, which is based on the powerful Lucene search toolbox, is a separate project in the Hibernate family and aims to provide a JPA "feeling" in its API as it also incorporates a lot of JPA interfaces in its codebase. 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).

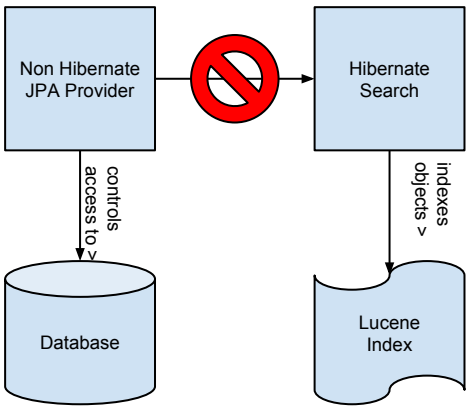

Figure 2: Hibernate Search's incompatibility with other JPA implementations

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 implementations might be of strategic importance, for others it could just be sheer preference to use a different JPA implementation.

Currently, developers that do not want to bind themselves to Hibernate ORM have to resort to using different full text search systems like native Lucene[4], ElasticSearch[5] or Solr[6]. While this is always a viable option, for some applications Hibernate Search

[4]official Lucene website, see [17]
[5]ElasticSearch Java API, see [3]
[6]Solr Java API, see [4]

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 [7], we can 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 most of the codebase could be reused for a generic version of Hibernate Search.

Creating this would be a better approach for a search API on top of JPA rather than rewriting everything from scratch. Hibernate Search could then act as the standard for fulltext search in the JPA world instead of having a competing API that would just do the same thing in a different style.

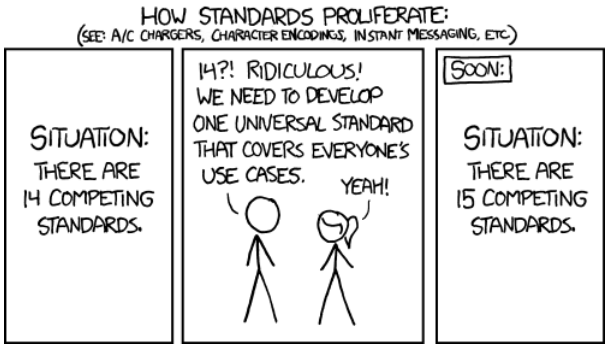

Figure 3: xkcd.com on competing standards [8]

This is why in this thesis we will show how such a generic version can be built. First, 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.

[7] Hibernate Search GitHub repository, see [12]
[8] xkcd comic #927, see [15]

2 Overview of technologies

Before we start going into detail about how to work with Hibernate Search in a generic environment, we will give a short overview of relevant technologies first. We will explain why ORMs in general and the JPA specification in particular are beneficial. Then, we will explain what fulltext search engines are used for and give a short overview about the available solutions for Java. We will see that generalizing Hibernate Search for any JPA implementation is a good approach and that it has benefits over using the different search solutions available.

2.1 Object Relational Mappers

Nowadays, many popular languages like Java, C#, etc. are object-oriented[9]. While SQL solutions for querying relational databases exist for these languages (JDBC for Java[10], OleDb for C#[11]), the user either has to work with the rowsets manually or convert them into custom data transfer objects (DTO) to gain at least some "real" objects to work with. Both approaches don't suit the object oriented paradigm well as SQL "flattens" the data into rows with when querying while a well designed class model would work with multiple classes in a hierarchy.

```
1 SELECT author.id, author.name, book.id, book.name
2 FROM author_book, author, author
3 WHERE author_book.bookid = book.id
4 AND author_book.authorid = author.id
```

Listing 1: sql query "flattening" the author and book table into rows

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. The returned objects are part of a complex object hierarchy and are easier to use from a object oriented point of view.

```
1 List<Author> data = orm.query("SELECT a FROM Author a " +
2         "LEFT OUTER JOIN a.books");
3 for(Author author : data) {
4         System.out.println("name: " + author.getName() +
5                 ", books: " + author.getBooks());
6 }
```

Listing 2: ORM query example

[9] Wikipedia on Object Oriented Programming (OOP), see [5]
[10] Oracle JDBC overview, see [13]
[11] OleDb usage page, see [14]

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), while the business logic would not changing a bit.

2.2 JPA

The first version of the JPA standard was released in May 2006. From then on it rose to being probably the most commonly used persistence API for Java and is considered the "industry standard approach for Object Relational Mapping"[12]. While mostly known for standardizing relational database mappers (ORM), it also supports other concepts like NoSQL[13] [14] or XML storage[15]. 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.[16].

Some popular relational implementations are:

- Hibernate ORM (JBoss)[17]
- EclipseLink (Eclipse foundation)[18]
- OpenJPA (Apache foundation)[19]

Using the standardized JPA API over any native ORM API has one really interesting benefit: The specific JPA implementation can be swapped out as it comes with standards for many common use cases.

This is particularily 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).[20] Many Java EE Web-servers ship with a bundled JPA implementation that they are optimized for (Wildfly with Hibernate ORM, GlassFish with EclipseLink, ...). This means that if the server is switched, it could also be a reasonable idea 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.

[12]Wikibooks on Java Persistence, see [6]
[13]Hibernate OGM project homepage, see [7]
[14]EclipseLink project homepage, see [11]
[15]EclipseLink project homepage, see [11]
[16]Wikipedia on Java Persistence API, see [1]
[17]Hibernate ORM project homepage, see [8]
[18]EclipseLink project homepage, see [11]
[19]OpenJPA project homepage, see [9]
[20]Wikipedia on Java EE, see [16]

2.3 Fulltext search engines

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 [21]:

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

Listing 3: SQL LIKE operator in use

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 would use vendor-specific features not every RDBMS supports.

Fulltext search engines can be used to complement databases in this regard. They are generally not intended to be replacing the database, but 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 focusing on their usage, features and compatibility with the JPA standard.

[21] w3schools on SQL LIKE, see [10]

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.[22]

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

2.3.1.1 Concepts As Lucene's focus is not on storing relational data, it comes with its own set of concepts. Following is a short overview over the concepts it has. These are not only the basis for Lucene, but also for the other search engines we will discuss later as they are based on Lucene's rich set of features.

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 (terms) are mapped to the texts they occur in[23], so it can be compared to a $Map < String, List < Text >>$ in Java. Before anything can be searched using Lucene, it has to be added to the the index (indexed) first.

Documents Documents are the data-structure Lucene stores and retrieves from the index. An index can contain zero or more Documents.

Fields A Document consists of at least one field. Fields are basically tuples of key and value. They can be stored (retrievable from the index) and/or indexed (used for searches, generate hits).

Analyzers Before documents get indexed, their fields are analyzed with one of the many Analyzers first. Analysis is the process of modifying the input in a manner such that it can be searched upon (stemming, tokenization, ...).

2.3.1.2 Usage Using Lucene as a standalone requires the programmer to design the engine from the bottom up. The developer has to write all the logic, starting with the actual index writing control mechanism, and the conversion code from Java objects to Documents, through to the index searcher control, and the query code with the conversion from Documents back to Java objects. This whole process requires a lot of code to be written and the API just helps by providing the necessary tools. This has

[22] official Lucene website, see [17]
[23] Lucene basic concepts, see [18]

one additional problem though: The Lucene API tends to change a lot between versions and the code has to be kept up-to-date. It's not uncommon that whole features that worked in one version are deprecated (potentially unstable, marked to be removed in the future) in the next release, resulting in big code changes being potentially necessary.

2.3.1.3 Features Lucene probably is the most complete toolbox to build a search-engine from. It has pre-built analyzers for many languages, a queryparser to support user written queries, a phonetic module, a faceting module, and many more features. First starting as a text only search engine, it has had support for spatial indexing for some while now. This and the support for faceting means that it is increasingly becoming more of a general search toolbox.

2.3.1.4 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 as it doesn't even have a mapper to convert from objects to its document representation and back. 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 however, this has to be done completely manually as it natively doesn't have any integration with databases.

2.3.2 Solr

> Solr is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene™.

2.3.3 ElasticSearch

2.3.4 Hibernate Search

> Hibernate Search transparently indexes your objects and offers fast regular, full-text and geolocation search. Ease of use and easy clustering are core.[24]

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 Reasoning of decision for Hibernate Search

[24] Hibernate Search project homepage, see [2]

3 Challenges

While building the generic version of Hibernate Search, we will encounter some challenges. We will now discuss the biggest ones and introduce a small example project. This project will be used to showcase some problems and usages later on in this thesis as well.

3.1 The example project

Consider a software built with JPA that is used to manage the inventory of a bookstore. It stores information about the available books (ISBN, title, genre, short summary of the contents) and the corresponding authors (surrogate id, first & last name, country) in a relational database. Each author is related to zero or more Books and each Book is written by one or more Authors. The entity relationship model diagram defining the database looks like this:

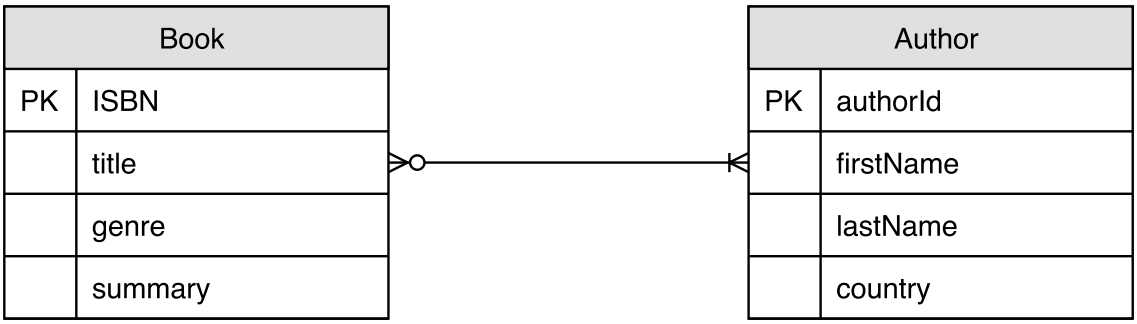

Figure 4: the bookstore entity relationship model

Using a mapping table for the M:N relationship of Author and Book, the database contains three tables: Author, Book and Author_Book. The JPA annotated classes for these entities are defined as following:

```
1  @Entity
2  @Table(name = "Book")
3  public class Book {
4
5          @Id
6          @Column(name = "isbn")
7          private String isbn;
8
9          @Column(name = "title")
10         private String title;
11
12         @Column(name = "genre")
13         private String genre;
```

```
14
15          @Lob
16          @Column(name = "summary")
17          private String summary;
18
19          @ManyToMany(mappedBy = "books", cascade = {
20                  CascadeType.MERGE,
21                  CascadeType.DETACH,
22                  CascadeType.PERSIST,
23                  CascadeType.REFRESH
24          })
25          private Set<Author> authors;
26
27          //getters & setters ...
28  }
```

Listing 4: Book.java

```
1  @Entity
2  @Table(name = "Author")
3  public class Author {
4
5          @Id
6          @GeneratedValue(strategy = GenerationType.AUTO)
7          @Column(name = "authorId")
8          private Long authorId;
9
10         @Column(name = "firstName")
11         private String firstName;
12
13         @Column(name = "lastName")
14         private String lastName;
15
16         @Column(name = "country")
17         private String country;
18
19         @ManyToMany(cascade = {
20                 CascadeType.MERGE,
21                 CascadeType.DETACH,
22                 CascadeType.PERSIST,
23                 CascadeType.REFRESH
24         })
25         @JoinTable(name = "Author_Book",
26                 joinColumns =
27                         @JoinColumn(name = "authorFk",
28                                 referencedColumnName = "authorId"),
29                 inverseJoinColumns =
30                         @JoinColumn(name = "bookFk",
31                                 referencedColumnName = "isbn"))
```

```
32          private Set<Book> books;
33
34          //getters & setters ...
35  }
```

Listing 5: Author.java

For the sake of simplicity and since every JPA provider is able to derive a default DDL script from the annotations, we don't supply any information about how to create the schema here. However, for real world applications defining a hand-written DDL script might be a better idea since the generated code might not be optimal and differs between the different JPA implementations and RBDMSs used.

3.2 Indexing & searching

Hibernate Search's engine wasn't designed to be used directly by application developers. Its main purpose is to serve as an integration point for other APIs that need to leverage its power to index object graphs and query the index for hits. This is why we have to write our own standalone module based on the "hibernate-search-engine" to ease its general usage. After the standalone is finished, we will build an integration of it with JPA to mimic the usage of Hibernate Search ORM as good as possible. By incorporating the same engine that the original does, we keep almost all of the indexing behaviour and even stay compatible with entities designed for it.

3.3 Index rebuilding

If the way objects are indexed changes, the existing files have to be purged and recreated in the new index format. The naive approach here would be purging the index and then indexing all data sequentially as they are retrieved from the database:

```
1  EntityManager em = ...;
2  <Hibernate Search Controller> search = ...;
3
4  search.purgeAll(Book.class);
5
6  Query query = em.createQuery("SELECT b FROM Book b");
7  List<Book> booksFromDb = query.getResultList();
8  for(Book b : booksFromDb) {
9          search.index(b);
10 }
```

Listing 6: naive index rebuilding

While this might work for small databases, bigger datasets will cause this algorithm to run out of memory, since we just retrieve all the data at once. This could be fixed by

implementing a batching strategy, but it would still be quite slow as it only uses one thread which would mostly be used for I/O from the database.

This is not optimal, since a index rebuild should be as fast as possible as the application cannot be properly used while the job is running. Therefore we need to create a parallel indexing mechanism, just like the one Hibernate Search ORM has.

3.4 Automatic index updating

The most important feature to be re-built, is automatic index updating. In Hibernate Search ORM, every change in the database is automatically reflected in the index. It is important to have this feature, because otherwise developers would have to manually make sure the index is always up-to-date. With bigger project sizes it gets increasingly harder to keep track of all the locations in the code that change index relevant data and inconsistencies in the indexing logic become nearly unavoidable. While this problem might be mitigated by hiding all the database access logic behind a service layer, even such a solution would be hard to keep error-free as for big applications this layer will probably have multiple critical indexing relevant spots as well.

The original Hibernate Search ORM is achieving an up-to-date index by listening to specific Hibernate ORM events for all of the C_UD (CREATE, UPDATE DELETE) actions. These events also cover entity relationship collections (for example represented by mapping tables like Author_Book). As our goal is to create a generic Hibernate Search engine that works with any JPA implementation, we cannot rely on any vendor specific event system. Thus, a different solution has to be found.

Hier evtl. noch die verschiedenen Möglichkeiten vorstellen? Eigentlich gehören die ja doch später in ihr eigenes Kapitel oder nicht?

4 indexing & searching

4.1 Using Hibernate Search's engine

4.2 Standalone version of Hibernate Search

4.3 Standalone integration with JPA interfaces

5 index rebuilding

6 automatic index updating

References

[1] Wikipedia https://en.wikipedia.org/wiki/Java_Persistence_API, 07/16/2015

[2] Hibernate Search project homepage http://hibernate.org/search/, 07/26/2015

[3] ElasticSearch Java API [https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html], 07/27/2015

[4] Solr Java API https://wiki.apache.org/solr/Solrj, 07/27/2015

[5] Wikipedia on Object Oriented Programming (OOP) https://en.wikipedia.org/wiki/Object-oriented_programming, 07/27/2015

[6] Wikibooks on Java Persistence https://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F, 07/27/2015

[7] Hibernate OGM project homepage http://hibernate.org/ogm/, 07/27/2015

[8] Hibernate ORM project homepage http://hibernate.org/orm/, 07/27/2015

[9] OpenJPA project homepage http://openjpa.apache.org/, 07/27/2015

[10] w3schools on SQL LIKE http://www.w3schools.com/sql/sql_like.asp, 07/27/2015

[11] EclipseLink project homepage http://www.eclipse.org/eclipselink/, 07/27/2015

[12] Hibernate Search GitHub repository https://github.com/hibernate/hibernate-search, 07/26/2015

[13] Oracle JDBC overview http://www.oracle.com/technetwork/java/javase/jdbc/index.html, 07/27/2015

[14] Documentation on how to use OleDb with .NET https://msdn.microsoft.com/en-us/library/5ybdbtte(v=vs.71).aspx, 07/27/2015

[15] xkcd #927 on competing standards https://xkcd.com/927/, 07/26/2015

[16] Java Platform, Enterprise Edition Wikipedia https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition, 07/16/2015

[17] Lucene Website https://lucene.apache.org/core/, 07/16/2015

[18] Lucene Tutorial http://www.lucenetutorial.com/basic-concepts.html, 07/20/2015

Appendix

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.

$Unterschrift$: $Ort, Datum$: