

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 26, 2015

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

## **Zusammenfassung**

## **Abstract**

# Contents

<b>1</b>	<b>Preface</b>	<b>4</b>
<b>2</b>	<b>Challenges</b>	<b>7</b>
2.1	The example project . . . . .	7
2.2	indexing & searching . . . . .	9
2.3	index rebuilding . . . . .	9
2.4	automatic index updating . . . . .	10
<b>3</b>	<b>Overview of technologies</b>	<b>11</b>
3.1	Object Relational Mappers . . . . .	11
3.2	JPA . . . . .	11
3.3	Fulltext search engines . . . . .	12
3.3.1	Lucene . . . . .	12
3.3.2	Solr . . . . .	14
3.3.3	ElasticSearch . . . . .	14
3.3.4	Hibernate Search . . . . .	14
<b>4</b>	<b>indexing &amp; searching</b>	<b>15</b>
4.1	Using Hibernate Search's engine . . . . .	15
4.2	Standalone version of Hibernate Search . . . . .	15
4.3	Standalone integration with JPA interfaces . . . . .	15
<b>5</b>	<b>index rebuilding</b>	<b>16</b>
<b>6</b>	<b>automatic index updating</b>	<b>17</b>
	<b>Literaturverzeichnis</b>	<b>18</b>
	<b>References</b>	<b>18</b>
	<b>Anhang</b>	<b>19</b>
	<b>Eidesstattliche Erklärung</b>	<b>19</b>

# 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 <sup>1</sup>, 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.<sup>2</sup>

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.

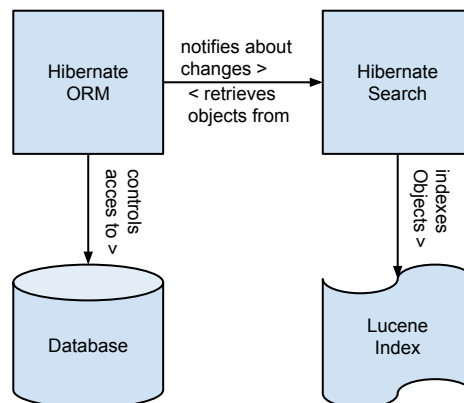


Figure 1: Hibernate Search with Hibernate ORM

<sup>1</sup>Wikipedia on Java Persistence API, see [1]

<sup>2</sup>Hibernate Search project homepage, see [2]

Hibernate Search, which is based on the powerful Lucene search toolbox, is a separate project in the Hibernate family and is using 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).

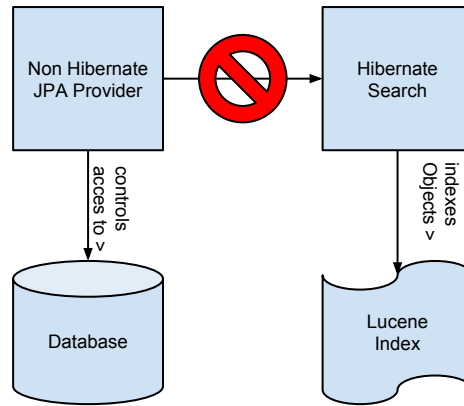


Figure 2: Hibernate Search's incompatibility with other JPA implementations

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 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, 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 <sup>3</sup>, 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.

<sup>3</sup>Hibernate Search GitHub repository, see [3]

Creating this would be a better approach for a search API on top of JPA 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.

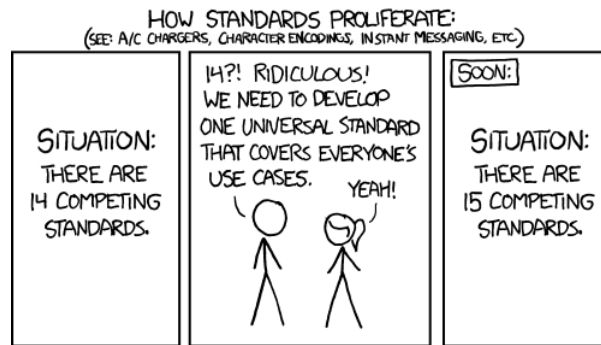


Figure 3: xkcd.com on competing standards <sup>4</sup>

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.

---

<sup>4</sup>xkcd comic #927, see [4]

## 2 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.

### 2.1 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:

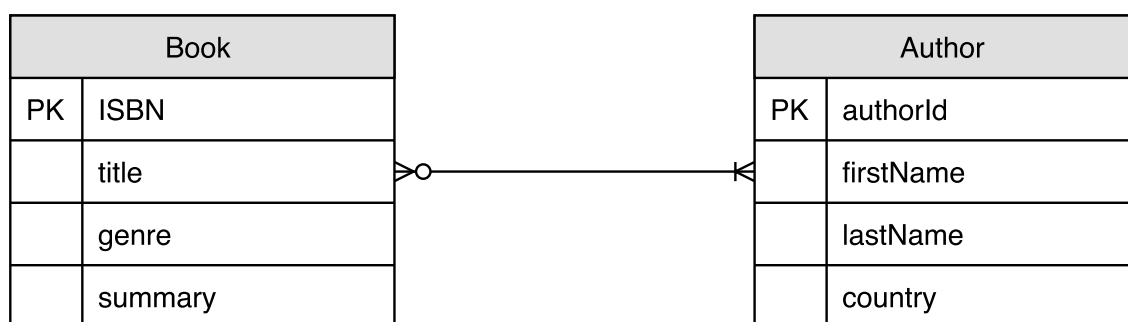


Figure 4: the bookstore entity relationship model

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 1: 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 2: 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 tables 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 RDBMS used.

## 2.2 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.

## 2.3 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 3: 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 Hibernate Search ORM has one.

## 2.4 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?

## 3 Overview of technologies

### 3.1 Object Relational Mappers

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 for at least some amount of object oriented style. 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.

### 3.2 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. 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.<sup>5</sup>

Some popular relational implementations are:

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

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).<sup>6</sup> 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

---

<sup>5</sup>Wikipedia on Java Persistence API, see [1]

<sup>6</sup>Wikipedia on Java EE, see [5]

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.

### 3.3 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:

```
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 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.

#### 3.3.1 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.<sup>7</sup>

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.

---

<sup>7</sup>official Lucene website, see [6]

**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.<sup>8</sup> Also, before anything can be searched using Lucene, it has to be added to the index first.

**Concepts** Lucene has its own set of concepts that need to be discussed first before we can take a look at its 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,

**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

---

<sup>8</sup>Lucene basic concepts, see [7]

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.

### **3.3.2 Solr**

### **3.3.3 ElasticSearch**

### **3.3.4 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?

## **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](https://en.wikipedia.org/wiki/Java_Persistence_API), 07/16/2015
- [2] Hibernate Search project homepage <http://hibernate.org/search/>, 07/26/2015
- [3] Hibernate Search GitHub repository <https://github.com/hibernate/hibernate-search>, 07/26/2015
- [4] xkcd #927 on competing standards <https://xkcd.com/927/>, 07/26/2015
- [5] Java Platform, Enterprise Edition Wikipedia [https://en.wikipedia.org/wiki/Java\\_Platform,\\_Enterprise\\_Edition](https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition), 07/16/2015
- [6] Lucene Website <https://lucene.apache.org/core/>, 07/16/2015
- [7] 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 :*

