



UNIVERSITÄT
BAYREUTH



University of Bayreuth
Institute for Computer Science

Bachelor Thesis

in Computer Science

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

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

Version date: September 8, 2015

1. Supervisor: Prof. Dr. Stefan Jablonski
2. Supervisor: Prof. Dr. Bernhard Westfechtel

Abstract

Fulltext search engines are a powerful tool to improve query results in applications where relational databases don't suffice. However, they don't integrate well with the widely spread concept of object relationship mappers (ORM, in Java predominantly represented by the standard JPA) in the object oriented programming world.

This is where Hibernate Search comes into use for Java developers: It combines JPA and fulltext search by being the intermediary between Hibernate ORM and a Lucene based fulltext index. It has one problem though: Hibernate Search only works with Hibernate ORM but not with other JPA-conform providers even though it is possible to support these. In this thesis we will show how such a generic version can be accomplished.

After discussing the methods we use, we give an explanation why a generic Hibernate Search is a desirable solution for JPA developers. Creating it is challenging as we have to build a standalone version of Hibernate Search's internal engine first and then integrate it with JPA together with an automatic index updating mechanism. We solve these challenges and give a usage example of the completed generic version. Finally, we discuss the current development state of the generic version and give an outlook on the planned merging process with the original Hibernate Search.

Zusammenfassung

Volltextsuchengines sind ein wertvolles Werkzeug um Suchergebnisse in Anwendungen zu verbessern, wenn relationale Datenbanken nicht ausreichen. Diese Engines sind jedoch nicht gut mit dem in der objekt-orientierten Programmierungs-Welt weit verbreiteten Konzept der Objekt-Relationalen Mapper (ORM, in Java vor allem durch den Standard JPA repräsentiert) integriert.

Für Java Entwickler bietet hier Hibernate Search eine Abhilfe: Es kombiniert JPA und Volltextsuche und stellt die Schnittstelle zwischen Hibernate ORM und einem Lucene basierten Volltextindex dar. Es hat aber ein Problem: Hibernate Search funktioniert nur in Kombination mit Hibernate ORM aber nicht mit anderen JPA konformen Providern, obwohl es möglich wäre diese zu unterstützen. In dieser Thesis wird daher gezeigt, wie eine solche generische Version realisiert werden kann.

Nachdem die benutzten Methoden erklärt wurden, wird eine Begründung dafür gegeben, warum Hibernate Search eine wünschenswerte Lösung für JPA Entwickler ist. Diese zu entwickeln ist eine Herausforderung, da wir zuerst eine Standalone Version von Hibernate Search's interner Engine bauen müssen um diese danach in eine JPA Version zusammen mit einem automatischen Index Updating Mechanismus zu integrieren. Wir zeigen, wie diese Probleme gelöst werden und erklären die Benutzung anhand eines Beispiels. Zuletzt gehen wir auf den aktuellen Entwicklungsstand der generischen Version ein und geben einen Ausblick auf den geplanten Merge-Prozess mit dem originalen Hibernate Search.

Contents

1	Preface	6
2	Methods	10
3	Overview of technologies	12
3.1	Object Relational Mappers	12
3.2	JPA	13
3.3	Fulltext search engines	14
3.3.1	Lucene	15
3.3.1.1	Concepts	15
3.3.1.2	Usage	17
3.3.1.3	Features	17
3.3.2	Fulltext search servers: ElasticSearch and Solr	18
3.3.2.1	Usage	18
3.3.2.2	Features	18
3.3.3	Hibernate Search	19
3.3.3.1	Usage	19
3.3.3.2	Features	19
3.3.4	Why a generic Hibernate Search?	20
4	Challenges	22
4.1	The example project	22
4.2	Standalone version	25
4.3	JPA integration	25
4.4	Automatic index updating	26
4.5	Timeline	26
5	Standalone version of Hibernate Search	28
5.1	Example project with Hibernate Search annotations	29
5.2	Usage of Hibernate Search's engine	32
5.2.1	Startup	32
5.2.2	Index manipulation	33
5.2.3	Queries	35
5.3	Design of the standalone version	38
5.3.1	Startup	40
5.3.2	Index manipulation	41
5.3.3	Queries	42
6	JPA integration of the standalone version	44
6.1	Architecture of Hibernate Search ORM	45
6.2	Startup	46
6.2.1	Index manipulation	47
6.2.2	Queries	48
6.2.3	Index rebuilds	49
6.3	Architecture of the generic version	50
6.3.1	Startup	51
6.3.2	Index manipulation	53

6.3.3	Queries	56
6.3.4	Index rebuilds	57
7	Automatic index updating	58
7.1	Description of different implementations	59
7.1.1	Synchronous approach	60
7.1.1.1	JPA events	60
7.1.1.2	Native integration with JPA providers	63
7.1.2	Asynchronous approach	64
7.1.2.1	Trigger architecture	65
7.1.2.2	Table creation	66
7.1.2.3	Event retrieval	70
7.2	Comparison of approaches	72
7.2.1	Additional work	72
7.2.2	Features	73
7.2.3	Summary	74
8	Usage of Hibernate Search GenericJPA	76
8.1	Dependencies	76
8.2	Entities	77
8.3	persistence.xml	80
8.4	Code usage example	82
9	Outlook	84
	Used software	86
	Listings	88
	Tables	102
	References	104
	Erklärung	108

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 of lesser importance to the average developer compared to the actual business logic 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 providing a standardized API for ORMs. First conceived in 2006 as part of EJB 3.0 ^{1 2}, 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 the standardized query API and are 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. For example, some ship with additional modules to enhance their capabilities. A perfect example for this is the Hibernate Search API aimed at Hibernate ORM users.³

4

¹JSR 220: Enterprise Java Beans 3.0, see [1]

²Javaworld: Understanding JPA, Part 1, see [2]

³Hibernate ORM project homepage, see [3]

⁴Hibernate Search project homepage, see [4]

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 as shown in figure 1: It works atop the Hibernate ORM, a popular JPA implementation, 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.

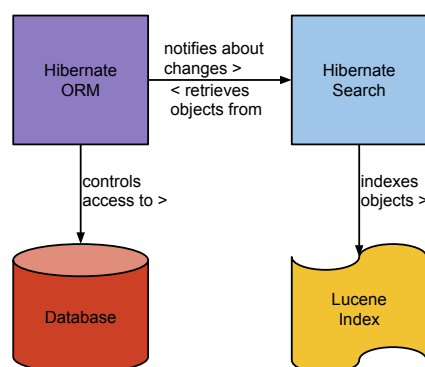


Figure 1: Hibernate Search with Hibernate ORM

Hibernate Search is based on the powerful Lucene search toolbox ⁵ ⁶ and is a separate project in the Hibernate family. It 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) as the following figure 2 shows.

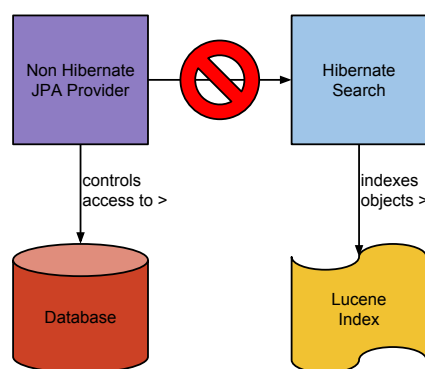


Figure 2: Hibernate Search's incompatibility with other JPA implementations

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.

⁵sourcecode on Hibernate Search GitHub, see [5]

⁶Hibernate Search FAQ, see [6]

Currently, developers that do not want to bind themselves to Hibernate ORM have to resort to using different full text search systems like native Lucene⁷, ElasticSearch⁸ or Solr⁹. While this is always a viable option, Hibernate Search would be a much better suit for some applications because of its design with a entity structure in mind combined with the automatic index updating feature, if it just were compatible with generic JPA.

When investigating Hibernate Search's project structure¹⁰, we can see that "hibernate-search-orm" is the only module apart from some server-integration modules that depends on any ORM logic. The modules that contain the indexing engine, the replication logic, alternative backends, etc. are completely independent from it. This means, that most of the codebase could be reused for a generic version of Hibernate Search.

Creating such a generic Hibernate Search is a better approach for a search API on top of JPA rather than rewriting a JPA binding from scratch. Hibernate Search could then act as the de-facto 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 standards¹¹

This is why we will show how such a generic version can be built in this thesis. 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 together with an automated index updating mechanism.

⁷official Lucene website, see [7]

⁸ElasticSearch Java API, see [8]

⁹Solr Java API, see [9]

¹⁰Hibernate Search GitHub repository, see [5]

¹¹xkcd comic #927, see [10]

Short overview of contents:

In chapter 2 we explain the methods we use to build Hibernate Search GenericJPA. In chapter 3 we give an overview of the relevant technologies used in this thesis and give short introductions to several fulltext search engines and the reasoning behind Hibernate Search GenericJPA. In chapter 4 we introduce a small example project and explain the main challenges while developing Hibernate Search GenericJPA. In chapter 5 we describe the standalone version of Hibernate Search. In chapter 6 we explain how the JPA integration of the standalone version is designed. In chapter 7 we work out an automatic index updating mechanism for Hibernate Search GenericJPA. In chapter 8 we give a full explanation of how to use Hibernate Search GenericJPA using the example from chapter 4. In chapter 9 we give a summary of what we have achieved in this thesis and describe further steps.

2 Methods

For the development of the generic version of Hibernate Search we use a combined approach of **top-down**¹² and **bottom-up**¹³ software development: After dividing the project into submodules (top-down) we develop the "building blocks" first and integrate them into bigger mechanisms (up until the sub-modules) as the project goes on (bottom-up). This way we stay flexible in the early stages of development and only have to write "wiring code" in the later stages.

After having identified the "building blocks" we follow this process to achieve them:

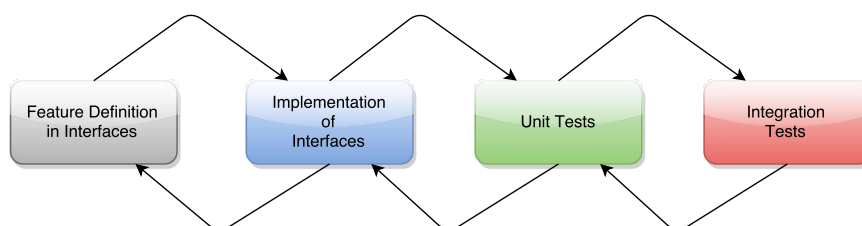


Figure 4: Development Process

- **Feature Definition in Interfaces:** We start by modelling the interfaces of our building blocks. While doing so, we try to be as compliant to the **Single Responsibility Principle**¹⁴ as possible. It helps by enforcing structures that are easy to reuse and change. However, we intentionally break it in some cases to allow more user-friendly interfaces (mostly in API entry-points).

By defining the features in interfaces and writing logic against only them (instead of the direct implementations), we achieve complete independence between the implementing classes and are compliant to the **Open-Closed-Principle**¹⁵ internally ("Modules should be both open (for extension) and closed (for modification)"¹⁶). In combination with the Single Responsibility Principle this allows us to write more "pluggable" code.

- **Implementation of Interfaces:** Once the interfaces are properly defined, we write implementations for them according to the contracts set. As stated above, these classes are generally written against other interfaces internally instead of direct implementations.

¹²Top-down programming, Robert Strandh, see [11]

¹³Bottom-up programming, Robert Strandh, see [12]

¹⁴objectmentor.com: Article on Single Responsibility Principle, see [13]

¹⁵objectmentor.com: Article on Open-Closed-Principle, see [14]

¹⁶Object-Oriented Software Construction, Prentice Hall, 1988, Bertrand Meyer, see [15]

- **Unit Tests:** Each feature must have a corresponding unit test. These are necessary to test each implementation for the right behaviour (outputs and side-effects) and stability for at least one given input. They also help to identify bugs in the implementations.
- **Integration Tests:** While Unit-Tests check the behaviour of every *single* implementation, Integration Tests are used to cover the correct behaviour when used together with other parts of the project. With these tests we ensure all features interoperate properly with each other.

Note that once a step is processed, that doesn't mean its result is final. As we can see in the diagram, we can go back and forth between the different steps at will to adapt to specific implementation problems and other new problems that have not been covered before.

We choose this kind of on-the-fly structure because it suits the project best: We have to investigate different approaches before we can work out the real solution. Additionally, because "hibernate-search-engine" is an internal API, we have to be as flexible as possible with our development since some features of it can be different than what we might expect in the first place.

It is worth mentioning that all the tests are executed during each build to ensure no regression bugs occur. This is automatically managed by the Maven ¹⁷ build tool.

¹⁷Maven project homepage, see [16]

3 Overview of technologies

Before we can go into detail about how to work with Hibernate Search in a generic environment, we will give a short overview of the 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.

3.1 Object Relational Mappers

Nowadays, many popular languages like Java or C# are object oriented. While SQL solutions for querying relational databases exist for these languages (JDBC for Java¹⁸, OleDb for C#¹⁹), 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 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 , book
3 WHERE author_book.bookid = book.id
4 AND author_book.authorid = author.id
```

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

This is one of the points 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 object hierarchy and are easier to use from a object oriented point of view as even relations that were not included in a join can generally be re-queried automatically when needed.

```
1 List<Author> data = orm.query("SELECT a FROM Author a");
2 for(Author author : data) {
3     // we can still fetch the books without joining in the query
4     System.out.println("name: " + author.getName() +
5         ", books: " + author.getBooks());
6 }
```

Listing 2: ORM query example

¹⁸Oracle JDBC overview, see [17]

¹⁹OleDb usage page, see [18]

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 by another (provided the ORM supports the specific RDBMS), while the business logic would not change 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 and is considered the "industry standard approach for Object Relational Mapping"^{20 21}. While mostly known for standardizing relational database mappers (ORM), it also supports other concepts like NoSQL^{22 23} or XML storage²⁴. 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 2.1²⁵.

Some popular relational implementations are:

- Hibernate ORM (Red Hat)²⁶
- EclipseLink (Eclipse foundation)²⁷
- OpenJPA (Apache foundation)²⁸

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 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).²⁹ 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 encounter many problems related to this switch.

²⁰Wikibooks on Java Persistence, see [19]

²¹Stackoverflow JPA tag, see [20]

²²Hibernate OGM project homepage, see [21]

²³EclipseLink project homepage, see [22]

²⁴EclipseLink project homepage, see [22]

²⁵JSR 338: JPA 2.1 specification, see [23]

²⁶Hibernate ORM project homepage, see [3]

²⁷EclipseLink project homepage, see [22]

²⁸OpenJPA project homepage, see [24]

²⁹Java EE specification on oracle.com [25]

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 ³⁰:

```
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)
- stemming and language specific optimisations
- comprehensive synonym support

There may exist some RDBMS that support similar query-types, but in the context of using an ORM we would then lose the ability to switch databases because of the usage of vendor-specific features that 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 (including Hibernate Search) focusing on their usage and features. After that, we will give the reasoning behind why a **generic** Hibernate Search is preferable to the other solutions.

³⁰w3schools on SQL LIKE, see [26]

3.3.1 Lucene

Apache LuceneTM 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.³¹

Lucene serves as the basis for many 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. Its latest stable version as of now is 5.3.0³².

3.3.1.1 Concepts As Lucene's focus is not on storing relational data, it comes with its own set of concepts. Following is a short overview of the most important ones. These are not only the basis for Lucene, but also for the other search engines we will discuss next, 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 in³³, so it can be compared to a *Map* \langle *String*, *List* \langle *Text* \rangle \rangle 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 and generating hits).

Analyzers Before documents get indexed, their fields are analysed 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, ...).

³¹official Lucene website, see [7]

³²official Lucene website, see [7]

³³Lucene basic concepts, see [27]

Example index The following figure 5 shows how an inverted index schematically looks like in Lucene. On the left we can see three different documents containing an id and the two text fields "field1" and "field2". The inverted index that stores references to these documents can be seen on the right. It contains all the different terms (field & value) mapped to the id of the texts they are contained in. The values of these terms have been analysed before they were stored into the index as they only contain singular words instead of the original "sentences" from the left.

Documents		
id	field1	field2
1	fulltext search lucene	search
2	lucene search	java
3	fulltext java	fulltext lucene

Inverted Index		
Term		Occurences
Field	Value	
field1	fulltext	1,3
field1	search	1,2
field1	lucene	1,2
field1	java	3
field2	search	1
field2	java	2
field2	fulltext	3
field2	lucene	3

Figure 5: schematic inverted index

3.3.1.2 Usage Using Lucene as a standalone engine requires the programmer to design the engine from the bottom up. The developer has to write all the logic, starting with the actual indexing code through to the code managing access to the index. The conversion from Java objects to Documents (for indexing) and back (for searching) have to be implemented as well. This whole process requires a lot of code to be written and the API only helps by providing the necessary tools. This is particularly problematic as 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 were state-of-the-art 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.

3.3.1.3 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 generating queries out of user input, a phonetic module, a faceting module, and many other features. While mostly known for its fulltext capabilities, it also has modules used for other purposes, for example the spatial module that enables geo-location query support.

One benefit of its low-level API is that it can easily be extended with custom analyzers, query-types, etc, though. This is especially useful for more sophisticated search engines.

3.3.2 Fulltext search servers: Elasticsearch and Solr

Lucene is the basis for two of the most popular search servers available: Elasticsearch (by elastic)³⁴ and Solr (sister project of Lucene)³⁵. Their current stable versions are 1.7.1³⁶ and 5.3.0³⁷ respectively.

3.3.2.1 Usage As both Elasticsearch and Solr are standalone server applications, they have to be configured before they can be used similar to the process of setting up a RDBMS. As they don't ship with any authentication mechanism by default they also have to be secured before they are used in production^{38 39}. Index changes and queries are done via a REST-like API (among other options).

3.3.2.2 Features As Elasticsearch and Solr are built upon Lucene, they support the same basic features that Lucene does, but add additional indexing and searching functionality and come with their own stack of tools to ease their usage (index inspectors, load analyzers, ...^{40 41}). They are generally used because of their good clustering capabilities (distribution & replication) and are optimized for high throughput and scalability^{42 43}. As they are not running inside the client application (as a native Lucene implementation would) these kind of servers don't force the user to use a specific programming language (in our case a JVM based one like Java).

³⁴ElasticSearch Homepage, see [28]

³⁵Solr Homepage, see [29]

³⁶ElasticSearch Download website, see [30]

³⁷Solr Homepage, see [29]

³⁸Solr security, see [31]

³⁹elastic Shield (security for Elasticsearch), see [32]

⁴⁰Solr Administration (Core Specific Tools), see [33]

⁴¹ElasticHQ, see [34]

⁴²ElasticSearch: Life inside a cluster, see [35]

⁴³Solr: Introduction to Scaling and Distribution, see [36]

3.3.3 Hibernate Search

From the GitHub README of Hibernate Search:

Full text search engines like Apache Lucene are very powerful technologies to add efficient free text search capabilities to applications. However, Lucene suffers several mismatches when dealing with object domain models. Amongst other things indexes have to be kept up to date and mismatches between index structure and domain model as well as query mismatches have to be avoided.

Hibernate Search addresses these shortcomings - it indexes your domain model with the help of a few annotations, takes care of database/index synchronization and brings back regular [JPA] managed objects from free text queries. ⁴⁴

Hibernate Search's current stable version is 5.4.0.Final which is based on Lucene 4.10.4 ⁴⁵.

3.3.3.1 Usage Hibernate Search is used in the context of JPA compliant applications using Hibernate ORM. It can easily be used by adding it to the classpath and setting some configuration properties in the JPA persistence.xml. It integrates seamlessly with JPA interfaces.

3.3.3.2 Features Similar to Elasticsearch and Solr, Hibernate Search is built upon Lucene and has similar features regarding indexing, searching and clustering but it is designed to be used in a JPA environment: it indexes JPA entities and the queries return them again.

It is tightly coupled with Hibernate ORM: while an integration with JPA is existent, Hibernate Search doesn't allow other JPA implementations than Hibernate ORM to be used as it internally relies on its code.

For future versions the Hibernate Search team is planning on adding Elasticsearch and Solr as additional backends ⁴⁶ besides the already existing Lucene based backend and the optional Infinispan integration.

⁴⁴Hibernate Search GitHub README, see [5]

⁴⁵hibernate-search-engine on mvnrepository.org, see [37]

⁴⁶Hibernate Search roadmap, see [38]

3.3.4 Why a generic Hibernate Search?

For Hibernate ORM developers Hibernate Search is probably currently the easiest way to have fulltext search capabilities in their application. While the native Lucene backend might not be the perfect choice for some applications (because they want to share the index with applications written in e.g. C#), the planned ElasticSearch and Solr backends would make up for this in the future.

Developers using other JPA implementations like EclipseLink or OpenJPA currently don't have the option to use a similar API to Hibernate Search as the Compass project has been discontinued (last version: 2.2.0 from Apr 06, 2009 as of mvnrepository.org⁴⁷).

In order to create a fulltext engine integrated with generic JPA creating a separate solution similar to Hibernate Search wouldn't be beneficial as it would include a lot of work and would probably not get much recognition.

A generic version of Hibernate Search however would use (most of) the already existing interfaces and would require a lot less code for the same behaviour and features as nearly all of the important Lucene logic can be found in modules not having any notion of Hibernate ORM. In fact, the only module of Hibernate Search requiring Hibernate ORM is "hibernate-search-orm".

Ultimately this generic version of Hibernate Search could also inspire some remodelling of the original Hibernate Search to incorporate generic JPA, which could make Hibernate Search the de-facto standard for fulltext search for the complete JPA world.

Using Hibernate Search and turning it into a general standard is definitely better than writing everything from scratch and thus "reinventing the wheel".

⁴⁷see <http://mvnrepository.com/artifact/org.compass-project/compass/2.2.0>

4 Challenges

While building the generic version of Hibernate Search, we will encounter some challenges. First, we will introduce a small example project. We will then use this project to illustrate the biggest challenges. It will also be used to showcase some problems and usages later on in this thesis.

4.1 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:

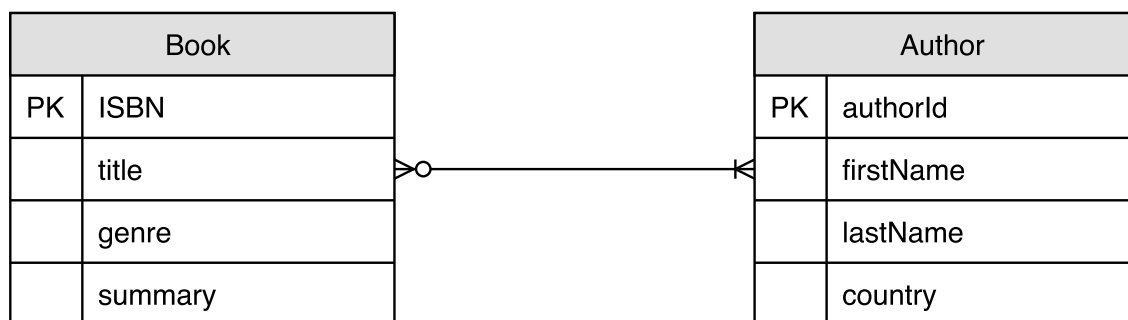


Figure 6: the bookstore entity relationship model

Using a mapping table for the M:N relationship of Author and Book, the database contains three tables: Author, Book and Author_Book. The applications strictly uses JPA to access the data without any vendor specific features. The JPA annotated classes for these entities are defined as shown in the following listings.

```
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 database 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 could differ between the different JPA implementations and RDBMSs used.

4.2 Standalone version

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 by exposing a quite low-level and in some ways complex API. This is why we have to write our own standalone version based on the "hibernate-search-engine" serving as an abstraction layer such that it eases the usage of the engine in our JPA integration.

4.3 JPA integration

After the standalone version is finished, we will build an integration of it with JPA. By incorporating the same engine that the original does, we will support the same indexing behaviour and even stay compatible with entities designed for the original with as little changes as possible. In fact, the main goal for the JPA integration is to be as compatible as possible with Hibernate Search ORM.

The implementations of these two challenges are represented by the modules "Hibernate Search Standalone" and "Hibernate Search GenericJPA" in the following figure 7. Together with the module "Hibernate Search Database Utilities", these are the sub-modules of our complete generic version and the result of the top-bottom analysis as described in chapter 2. Note that during this thesis we will be referring to the whole project by the name of the main module "Hibernate Search GenericJPA" as well.

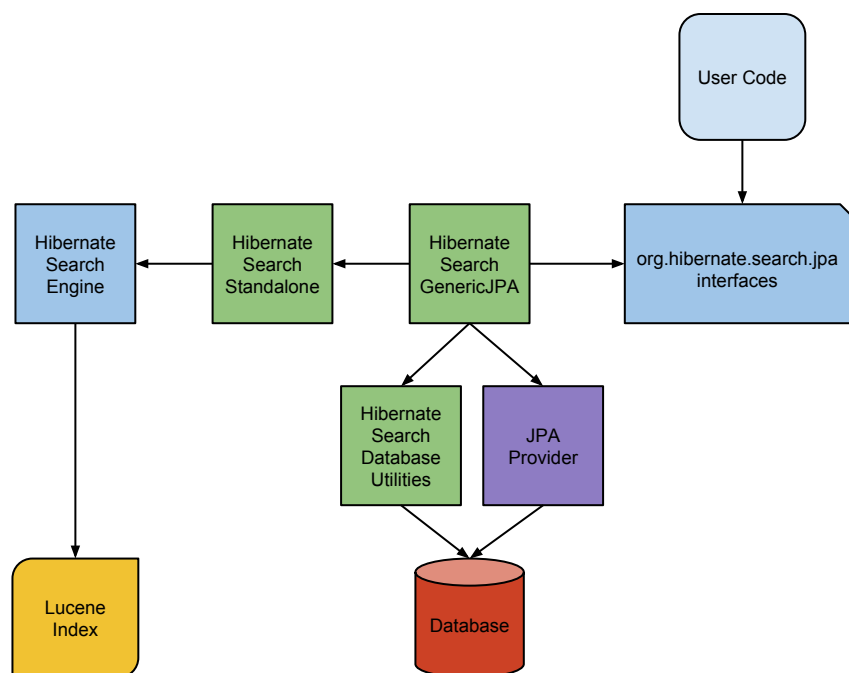


Figure 7: Complete Architecture of Hibernate Search GenericJPA

4.4 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, at least an additional generic solution has to be found.

This feature will be part of the "Hibernate Search GenericJPA" module.

4.5 Timeline

The solutions for the challenges depend on each other in the same order they were described above: the JPA integration can only be worked on as soon as the standalone integration is done and work on the automatic updating mechanism cannot be started without knowing the JPA integration interfaces. The timeline of our project therefore looks like this:

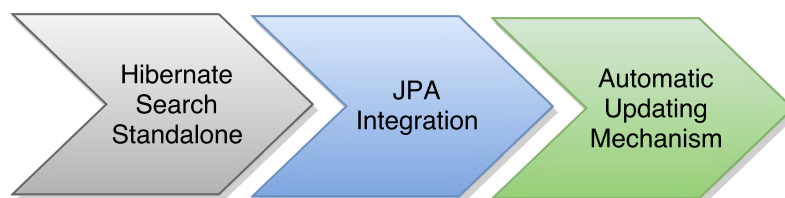
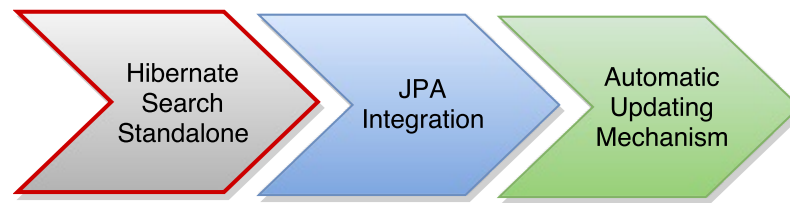


Figure 8: Timeline of the project

5 Standalone version of Hibernate Search



We will start the development part of this thesis by discussing how Hibernate Search's engine (in the form of the module "hibernate-search-engine") can be used in general. After this is done we will work out a standalone version of this engine that is easier to work with so we can integrate this standalone version with JPA in the next chapter.

As already described earlier in chapter 4.2, hibernate-search-engine is not intended to be used by application developers, but for other APIs to integrate with. Therefore there is no real public documentation available on how to use it besides the internal JavaDocs ⁴⁸ (describing the classes, but not the interaction between them). Nearly all the following information had to be retrieved from tests in the hibernate-search-engine and hibernate-search-orm integration module source code ⁴⁹.

⁴⁸Hibernate Search JavaDoc, see [48]

⁴⁹Hibernate Search GitHub, see [5]

5.1 Example project with Hibernate Search annotations

Before we explain how we do things in particular, we set up the example entities described in 4.1 as if the original Hibernate Search would have been used. We do so by adding additional annotations to our entity-classes (only the basic properties are explained here):

1. **@Indexed**: marks the entity as an index root-type.
2. **@DocumentId**: marks the field as the id of this entity. this is only needed if no JPA **@Id** can be found, but can be used to override settings. A Field marked with this is stored and indexed. Storing means that its contents are obtainable by projection when retrieving results. This is needed for ids so that the original Entity can be obtained from the database.
3. **@Field**: describes how the annotated field should be indexed:
@Field#store determines whether the contents of this Java property should be stored in the index (Store.YES) or not (Store.NO, default) while **@Field#index** determines whether it should be searchable in the index (Index.YES, default) or not (Index.NO).
The index fieldname defaults to the Java property name but can manually be overridden with **Field#name** if needed.
4. **@IndexedEmbedded**: marks properties that point to other classes which should be included in the index. By default, all fields contained in these entities are prefixed with the property name this is placed on.
@IndexedEmbedded#includeEmbeddedObjectId decides whether the ids of the embedded objects have to be stored and indexed as well.
5. **@ContainedIn**: used in entities that are embedded in other indexes. this is set on the properties that point back to the index-owning entity.

As these annotations are defined in `hibernate-search-engine`, we can rely on all of them while designing the standalone version of Hibernate Search and all other modules depending on it.

The resulting entities look like this:

```
1 @Entity
2 @Table(name = "Book")
3 @Indexed
4 public class Book {
5
6     @Id
7     @Column(name = "isbn")
8     @DocumentId
9     private String isbn;
10
11     @Column(name = "title")
12     @Field(store = Store.YES, index = Index.YES)
13     private String title;
14
15     @Column(name = "genre")
16     @Field(store = Store.YES, index = Index.YES)
17     private String genre;
18
19     @Lob
20     @Column(name = "summary")
21     @Field(store = Store.NO, index = Index.YES)
22     private String summary;
23
24     @ManyToMany(mappedBy = "books", cascade = {
25         CascadeType.MERGE,
26         CascadeType.DETACH,
27         CascadeType.PERSIST,
28         CascadeType.REFRESH
29     })
30     @IndexedEmbedded(includeEmbeddedObjectId = true)
31     private Set<Author> authors;
32
33     //getters & setters ...
34 }
```

Listing 6: Book.java with Hibernate Search annotations

```
1 @Entity
2 @Table(name = "Author")
3 public class Author {
4
5     @Id
6     @GeneratedValue(strategy = GenerationType.AUTO)
7     @Column(name = "authorId")
8     @DocumentId
9     private Long authorId;
10
11     @Column(name = "firstName")
12     @Field(store = Store.YES, index = Index.YES)
13     private String firstName;
14
15     @Column(name = "lastName")
16     @Field(store = Store.YES, index = Index.YES)
17     private String lastName;
18
19     @Column(name = "country")
20     @Field(store = Store.YES, index = Index.YES)
21     private String country;
22
23     @ManyToMany(cascade = {
24         CascadeType.MERGE,
25         CascadeType.DETACH,
26         CascadeType.PERSIST,
27         CascadeType.REFRESH
28     })
29     @JoinTable(name = "Author_Book",
30         joinColumns =
31             @JoinColumn(name = "authorFk",
32                 referencedColumnName = "authorId"),
33         inverseJoinColumns =
34             @JoinColumn(name = "bookFk",
35                 referencedColumnName = "isbn"))
36     @ContainedIn
37     private Set<Book> books;
38
39     //getters & setters ...
40 }
```

Listing 7: Author.java with Hibernate Search annotations

5.2 Usage of Hibernate Search's engine

In this chapter we will take a look at how to use Hibernate Search's engine natively by showing how it's started, how the index is manipulated and how searching works.

5.2.1 Startup

A Hibernate Search engine instance is represented by a **SearchIntegrator** object. In order to obtain it, we first have to write a special configuration class that implements **org.hibernate.search.cfg.spi.SearchConfiguration**. An object of this class has then to be created and filled with all the configuration properties Hibernate Search requires. The minimum that has to be set for this to work are the following:

1. **hibernate.search.default.directory_provider**: The two most common cases here are either "ram" or "filesystem". This decides where the index will be stored. A ram directory is only present in the system memory while the SearchIntegrator exists. A "filesystem" directory is persisted on the hard disk. For "filesystem" the additional property "hibernate.search.default.indexBase" has to be set to an appropriate path.
2. **hibernate.search.lucene_version**: This decides which Lucene version has to be used internally. The currently latest supported version supported is "5.2.1" as we are using an early alpha version of Hibernate Search for development (see "Used software" in the appendix). It can be set to earlier versions to support legacy behaviour in some Lucene classes.

A complete list of the available settings can be found in the Hibernate Search documentation ⁵⁰ (only the Hibernate ORM specific settings cannot be used). Our **StandaloneSearchConfiguration** (appendix listing 44) defaults to "ram" and "5.2.1".

Having this class in place, a **SearchIntegrator** can be obtained by a **SearchIntegratorBuilder** like this:

```

1 List<Class<?>> indexClasses = Arrays.asList(Book.class, Author.class);
2
3 SearchConfiguration searchConfiguration =
4     new StandaloneSearchConfiguration();
5 indexClasses.forEach( searchConfiguration::addClass );
6
7 //bootstrapping class for Hibernate Search
8 SearchIntegratorBuilder builder = new SearchIntegratorBuilder();
9

```

⁵⁰Hibernate Search documentation, see [39]


```
10 //we have to build an integrator here (the builder needs a
11 // "base integrator" first before we can add index classes)
12 builder.configuration( searchConfiguration ).buildSearchIntegrator();
13
14 indexClasses.forEach( builder::addClass );
15
16 //starts the engine with all configuration properties set
17 SearchIntegrator searchIntegrator = builder.buildSearchIntegrator();
18
19 //use the integrator ...
20
21 //close it
22 searchIntegrator.close();
```

Listing 8: Starting up the engine

5.2.2 Index manipulation

Now that we know how a `SearchIntegrator` can be built, we can take a look at how we can control the index using the engine's features.

The engine does a lot of optimizations in the backend. This is the reason the specifics are hidden behind a **Worker** pattern. Such a worker batches operations by synchronizing upon the `org.hibernate.search.backend.TransactionContext` interface. Our implementation of this is simply called **Transaction** (appendix listing 43). The different index operations are represented by **Work** objects that contain the `WorkType` (INDEX, UPDATE, PURGE, etc.) and all necessary data to execute the individual task.

Indexing objects with **WorkType.INDEX**:

```
1 Book book = ...;
2 Transaction tx = new Transaction();
3 Worker worker = searchIntegrator.getWorker();
4 worker.performWork( new Work( book, WorkType.INDEX ), tx );
5 tx.commit();
```

Listing 9: Indexing an object with the engine

Updating objects with **WorkType.UPDATE**:

```
1 Book book = ...;
2 Transaction tx = new Transaction();
3 Worker worker = searchIntegrator.getWorker();
4 worker.performWork( new Work( book, WorkType.UPDATE ), tx );
5 tx.commit();
```

Listing 10: Updating an object with the engine

Deleting objects with **WorkType.PURGE**:

```
1 String isbn = ...;
2 Transaction tx = new Transaction();
3 Worker worker = searchIntegrator.getWorker();
4 worker.performWork( new Work( Book.class, isbn, WorkType.PURGE ), tx );
5 tx.commit();
```

Listing 11: Deleting an object by id with the engine

This API doesn't have any "convenience" methods that wrap around the Transaction management if no batching is needed, nor does it have any wrapper utility for the Work object generation.

5.2.3 Queries

Querying the index is already acceptable to some extent when it comes to building the actual query. This is mainly due to the fact the query class **HSQuery** supports method chaining and that the same query builder DSL (which returns Lucene queries) used in Hibernate Search ORM is available. Any basic Lucene query could be used as well, but would require manual analysis of the input. Queries produced by the builder are automatically analysed with the correct Analyzer.

```
1 SearchIntegrator searchIntegrator = ...;
2
3 HSQuery query = searchIntegrator.createHSQuery();
4
5 //find information about all the entities matching a given title
6 List<EntityInfo> entityInfos =
7     query.luceneQuery(
8         //query DSL:
9         searchIntegrator.buildQueryBuilder()
10             .forEntity( Book.class )
11             .get()
12             .keyword()
13             .onField( "title" )
14             .matching( "searchString" )
15             .createQuery()
16     ).targetedEntities(
17         Collections.singletonList(
18             Book.class
19         )
20     ).projection(
21         ProjectionConstants.ID
22     ).queryEntityInfos();
```

Listing 12: Querying the index with the engine

Executing the queries doesn't return anything resembling the original Java objects, though. The actual data returned depends on what we project upon in the `projection(...)` call and is wrapped in an **EntityInfo** object. In our example we only retrieve the ids of the Books matching our query. We do this because when using a search index, we don't generally want to work with the actual data found in the index after the hits have been found. We want objects retrieved from the database.

```
1 //a JPA EntityManager
2 EntityManager em = ...;
3
4 //extract info from the entityInfos
5 for(EntityInfo entityInfo : entityInfos) {
6     String isbn = (String) entityInfo.getProjection()[0];
7     //retrieve an object from the database
8     Book book = em.find(Book.class, isbn);
9     //handle this information ...
10 }
```

Listing 13: Extracting info from the results

5.3 Design of the standalone version

In 5.2 we described how the engine can be used natively without any notion of JPA. While using the engine this way is possible, it is not convenient because some of the code is quite complicated. This is the reason we will now discuss a standalone abstraction of this code.

As we have seen in the examples earlier, the main interfaces used for index control and querying are **SearchIntegrator** and **HSQuery**. In order to abstract some of the complicated logic, we now introduce two new interfaces:

- **StandaloneSearchFactory**: This interface is responsible for all index changes. Code using this abstraction doesn't have to cope with the Worker pattern, at all. This is hidden behind index/delete/update methods.
- **HSearchQuery**: While still having the same chaining methods as **HSQuery**, we retrieve results from the index in a different manner now. Instead of manually having to extract the ID out of the **EntityInfos**, this interface retrieves the actually wanted data with the help of the **EntityProvider** interface which wraps the access to the database. The specifics of the **EntityProvider** are still use-case specific as the examples later in this chapter will show.

The following diagram shows the rough architecture of our new standalone version. Note that we are using a specialization of **SearchIntegrator** - namely **ExtendedSearchIntegrator** - which allows us to have more sophisticated features.

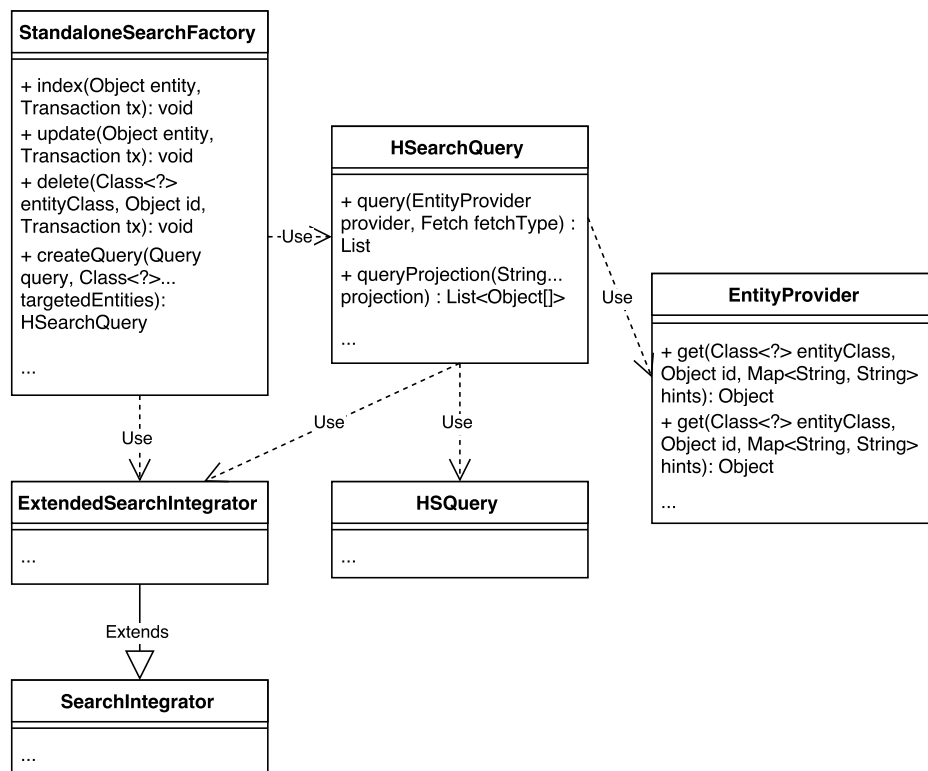


Figure 9: Rough architecture of the standalone version (important parts)

5.3.1 Startup

The startup process of the standalone version doesn't differ much from manually using the engine in terms of configuration as we still have to use the `SearchConfiguration` interface. The only difference is how we build the `StandaloneSearchFactory`. This is done with a **`StandaloneSearchFactoryFactory`**, so the code using it doesn't have to handle the creation of the actual implementation object.

```
1 List<Class<?>> indexClasses = Arrays.asList(Book.class, Author.class);
2
3 //we still have to build the SearchConfiguration object
4 SearchConfiguration searchConfiguration =
5     new StandaloneSearchConfiguration();
6 indexClasses.forEach( searchConfiguration::addClass );
7
8 //the builder pattern from before is abstracted in the following lines
9 StandaloneSearchFactory searchFactory =
10     StandaloneSearchFactoryFactory.
11         createSearchFactory(
12             searchConfiguration,
13             indexClasses
14         );
15
16 //use the searchfactory ...
17
18 //close it
19 searchFactory.close();
```

Listing 14: Starting up the standalone version

5.3.2 Index manipulation

With our standalone version, basic index control becomes more streamlined as we don't have to work with the SearchIntegrator's Worker pattern anymore as it was described in chapter 5.2.2:

```
1 Book book = ...;
2 Transaction tx = new Transaction();
3 searchFactory.index(book, tx);
4 tx.commit();
```

Listing 15: Indexing an object with the standalone version

```
1 Book book = ...;
2 Transaction tx = new Transaction();
3 searchFactory.update(book, tx);
4 tx.commit();
```

Listing 16: Updating an object with the standalone version

```
1 Transaction tx = new Transaction();
2 String isbn = ...;
3 searchFactory.delete(Book.class, isbn, tx);
4 tx.commit();
```

Listing 17: Deleting an object by id with the standalone version

5.3.3 Queries

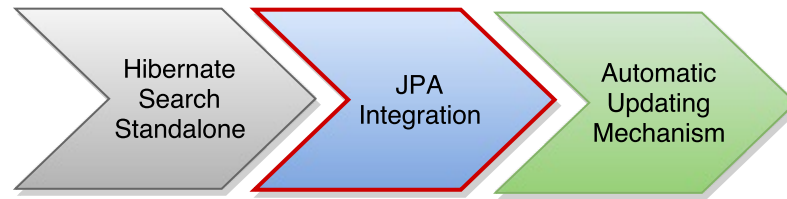
The biggest change in the standalone version is probably how the index is queried. We don't have to work with EntityInfos anymore as we introduced the **EntityProvider** interface. This interface hosts one method that is to be used for batch fetching (Fetch.BATCH) and one for single fetching (Fetch.FIND_BY_ID).

A good default implementation delegating the database access to a JPA EntityManager is our **BasicEntityProvider** (listing 45 in the appendix). Besides taking an EntityManager in its constructor, it also needs a Map<Class<?>, String> containing the id properties of the entities. While we leave the construction of this map out in the following listing 18 for the sake of simplicity, the code for this can be found in listing 46 in the appendix. After its creation, this map can then be stored in a central place and reused.

```
1 StandaloneSearchFactory searchFactory = ...;
2
3 EntityManager em = ...;
4 Map<Class<?>, String> idProperties = ...;
5
6 EntityProvider entityProvider = new BasicEntityProvider(em, idProperties);
7
8 List<Book> books = searchFactory
9     .createQuery(searchFactory.buildQueryBuilder()
10         .forEntity(Book.class)
11         .get()
12         .keyword()
13         .onField("title")
14         .matching("searchString")
15         .createQuery(), Book.class
16     ).query(
17         entityProvider,
18         Fetch.BATCH
19     );
```

Listing 18: Querying the index with the standalone version

6 JPA integration of the standalone version



After simplifying the access to Hibernate Search's engine we will work out an integration with JPA interfaces next. Since we started with the premise of not wanting to "reinvent the wheel" by writing everything from scratch (as described in 3.3.4), we will try to build an integration as similar to the JPA interfaces of Hibernate Search ORM as possible.

Before we can go into detail about how we build our integration, we have to discuss the general architecture first. We will go over how the Hibernate Search ORM integration with JPA interfaces behaves from a user's point of view and then take a look at what has to be changed in order to be compatible with any JPA implementor.

6.1 Architecture of Hibernate Search ORM

Hibernate Search ORM integrates with the JPA API by extending the interfaces `javax.persistence.EntityManager` and `javax.persistence.Query` and adds new functionality to the fulltext search versions of these interfaces: **FullTextEntityManager** and **FullTextQuery**. The following figure shows a rough overview of this. Note that this contains only the methods relevant for the following sections.

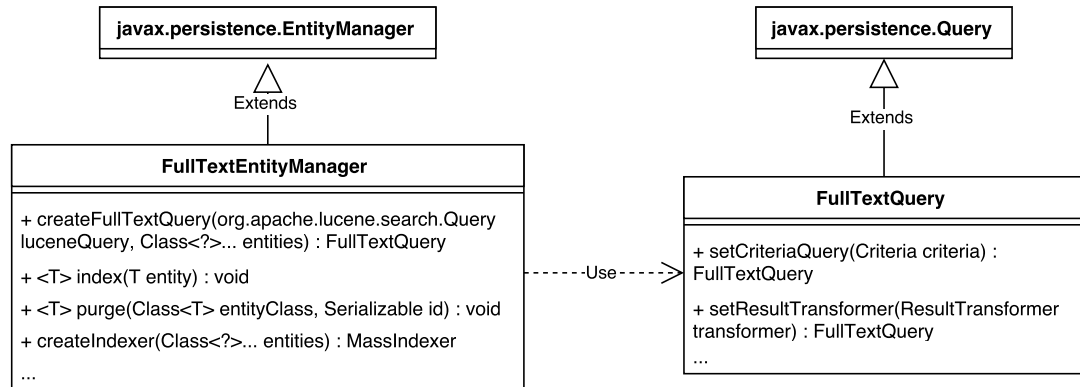


Figure 10: The main JPA interfaces of Hibernate Search ORM

6.2 Startup

As Hibernate Search ORM is tightly coupled with Hibernate ORM it is automatically started if found on the classpath and the persistence.xml contains the following:

```
1 ...  
2 <property name="hibernate.search.default.directory_provider" value="  
    filesystem"/>  
3 <property name="hibernate.search.default.indexBase" value="/path/to/  
    indexes"/>  
4 ...
```

Listing 19: Additions to persistence.xml with Hibernate Search ORM

This means that there exists no real code entry point as Hibernate Search is fully integrated into the Hibernate ORM/OGM lifecycle. FullTextEntityManagers can therefore be obtained with:

```
1 EntityManager em = ...;  
2 FullTextEntityManager fem = Search.getFullTextEntityManager(em);
```

Listing 20: Obtaining a FullTextEntityManager with Hibernate Search ORM

All of FullTextEntityManager's operations are controlled by the same transactions the original Hibernate EntityManager is using. This is the reason we will not have any search transaction related code in the following paragraphs.

6.2.1 Index manipulation

The index operations are all straightforward and similar to what we designed our standalone version in chapter 5.3 to work like apart from minor naming differences.

Hibernate Search ORM doesn't differentiate between indexing and updating.

```
1 FullTextEntityManager fem = ...;  
2 Book book = ...;  
3 fem.index(book);
```

Listing 21: Indexing/Updating an object with Hibernate Search ORM

Deleting objects from the index is called purging. This is probably due to not wanting to confuse it with JPA's delete(...).

```
1 FullTextEntityManager fem = ...;  
2 String isbn = ...;  
3 fem.purge(Book.class, isbn);
```

Listing 22: Deleting an object by id with Hibernate Search ORM

6.2.2 Queries

Hibernate Search ORM integrates even better with JPA for queries than our standalone version as the `FullTextQuery` interface extends the JPA `Query` interface and uses `getResultList()` to return its results:

```
1 EntityManager em = ...;
2 FullTextEntityManager fem = Search.getFullTextEntityManager(em);
3
4 FullTextQuery fullTextQuery = fem.createFullTextQuery(
5     fem.getSearchFactory().buildQueryBuilder()
6         .forEntity( Book.class )
7         .get()
8         .keyword()
9         .onField( "title" )
10        .matching( "searchString" )
11        .createQuery(),
12    Book.class);
13
14 List<Book> books = (List<Book>) fullTextQuery.getResultList();
```

Listing 23: Querying with Hibernate Search ORM

6.2.3 Index rebuilds

A noteworthy feature of Hibernate Search is its `MassIndexer`. It can be used whenever the way the entities are indexed is changed (e.g. in the `@Field` annotations). It uses multiple threads working in parallel to scroll results from the database and then indexes these efficiently. This is by far faster than the naive approach working in only one thread. It also incorporates a lot of internal improvements a normal developer wouldn't have access to as the specifics are hidden in the implementation packages of Hibernate Search which are not intended to be used outside of its own code.

A full index rebuild for our `Book` entity would look like this:

```
1 EntityManager em = ...;
2 FullTextEntityManager fem = Search.getFullTextEntityManager(em);
3
4 fem.createIndexer( Book.class )
5     .batchSizeToLoadObjects( 25 )
6     .threadsToLoadObjects( 12 )
7     .idFetchSize( 150 )
8     .transactionTimeout( 1800 )
9     .startAndWait();
```

Listing 24: `MassIndexer` usage with Hibernate Search ORM

"This will rebuild the index of all `[Book]` instances (and subtypes), and will create 12 parallel threads to load the `User` instances using batches of 25 objects per query; these same 12 threads will also need to process indexed embedded relations and custom `FieldBridges` or `ClassBridges`, to finally output a Lucene document."⁵¹

⁵¹Hibernate Search documentation (`MassIndexer`, v5.4), see [40]

6.3 Architecture of the generic version

As good as Hibernate Search ORM's API integration with JPA's EntityManager and Query interface is, its additional interfaces still contain some Hibernate ORM related features and logic that a generic version (we call it Hibernate Search GenericJPA) can not support and therefore have to be changed, emulated or removed altogether.

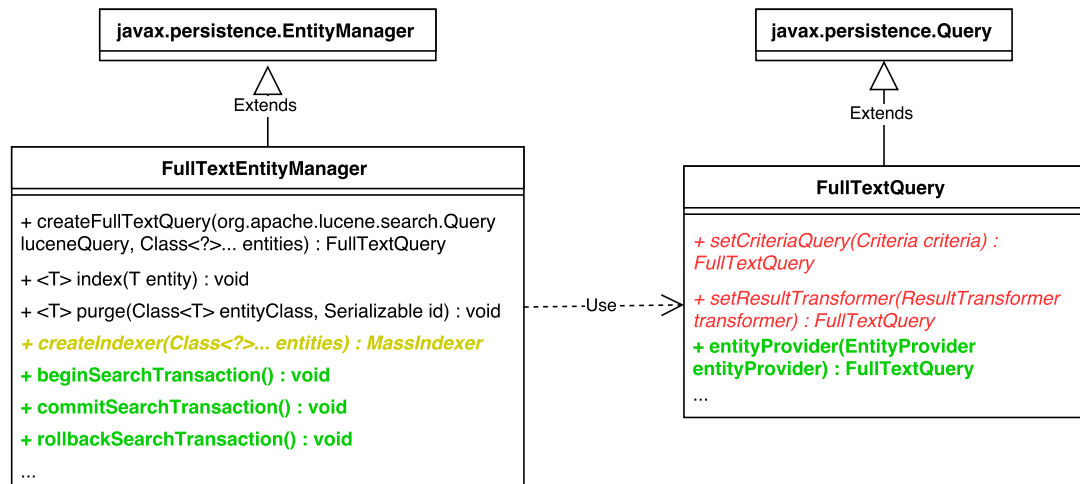


Figure 11: Required fixes for a generic version

In the figure 11 above, we have marked all the methods requiring to be fixed in the FullTextEntityManager and FullTextQuery interfaces:

- green: new methods
- red: methods that can't be supported
- olive: methods that can be supported if changed

Besides these, some other aspects need changes as well. We will describe the reasoning behind all of the needed changes & additions in the following paragraphs.

6.3.1 Startup

In our generic version we can't tightly integrate with the `EntityManagerFactory` of the JPA provider. This is the reason we introduce a separate interface called **JPA`SearchFactoryController`**:

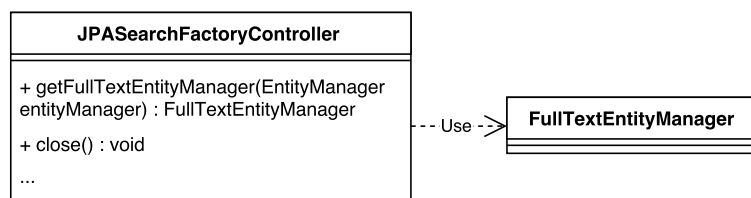


Figure 12: JPA`SearchFactoryController`

Having this separate interface means that the lifecycle of the generic version has to be controlled separately contrary to the standard Hibernate Search which is integrated with Hibernate ORM's lifecycle as described in chapter 6.2.

Unlike the static way a `FullTextEntityManager` is obtained in Hibernate Search ORM via the `Search` class, in our generic version, we obtain it with the **`getFullTextEntityManager(EntityManager entityManager)`** method (the `Search` class in Hibernate Search ORM only works because of the tight coupling of ORM and Search). This means that an instance of the `JPASearchFactoryController` has to be available at all times when access to the index is required.

Using a non-static approach here has one benefit, though: we can pass null to this method and get a search only `FullTextEntityManager` that can be used to work on the index when no database access is needed. This is particularly useful if POJOs have to be indexed which are not associated with JPA (see table 2, property "hibernate.search.additionalIndexedTypes").

We start the fulltext search engine with our bootstrapping class **Setup** like this:

```
1 //In Hibernate Search ORM, the fulltext engine would be started
2 //together with the EntityManagerFactory.
3 //In GenericJPA we can't do that.
4 EntityManagerFactory emf = ...;
5
6 EntityManager em = ...;
7
8 Properties properties = new Properties();
9
10 properties.setProperty(
11     "hibernate.search.searchfactory.type",
12     "manual-updates"
13 );
14
15 //In GenericJPA this starts the fulltext engine to
16 //which a reference is returned by this method call
17 JPASearchFactoryController searchFactoryController =
18     Setup.createSearchFactoryController( emf, properties );
19
20 //FullTextEntityManagers are not obtained with the Search class
21 FullTextEntityManager fem =
22     searchFactoryController.getFullTextEntityManager(em);
23
24 //use it...
25
26 searchFactoryController.close();
```

Listing 25: MassIndexer usage with Hibernate Search ORM

For this example we are using "manual-updates", as we haven't discussed how the index is kept up-to-date. After we worked that out, "manual-updates" will just be a fallback setting for developers not wanting to have the index automatically updated. Also note that there are many more properties that can be set and that vanilla Hibernate Search settings are passed this way as well. A complete list of the available GenericJPA configuration properties can be found in table 2 in the appendix.

6.3.2 Index manipulation

In Hibernate Search ORM, all manual index manipulations are synchronized with the EntityManager transaction lifecycle (index changes underly the JPA transaction system). In our generic approach we cannot do this as JPA doesn't have an extension point for this kind of usage. This is the reason we introduce the [**begin/commit/rollback**]SearchTransaction() methods in FullTextEntityManager. These have to be used to control the transaction lifecycle of all the index manipulation methods:

```
1 EntityManager em = ...;
2 JPASearchFactoryController searchFactoryController = ...;
3
4 FullTextEntityManager fem =
5     searchFactoryController.getFullTextEntityManager(em);
6
7 fem.beginSearchTransaction();
8 try {
9     //index or purge here
10    fem.commitSearchTransaction();
11 } catch (Exception e) {
12    fem.rollbackSearchTransaction();
13    throw e;
14 }
```

Listing 26: Index control with Hibernate Search GenericJPA

Because manual index changes are not needed frequently in normal applications, we don't restrict the usage of GenericJPA in application servers by a lot compared to the original Hibernate Search ORM by introducing our own search transaction management methods. In general, Hibernate Search transactions can not be compared with real RDBMS transactions anyways as it is allowed to write changes to the index without committing with flushToIndexes(). Changes applied in this manner can not be reverted by a rollback.

One additional problem with supporting indexing generic JPA entities is that some JPA providers don't return objects of the original entity class. For example, EclipseLink returns an object of an anonymous subclass of the original in which it hides away some utility logic needed for lazy loading.

This is problematic because the engine needs to know which class to get the index description metamodel from. Therefore we have to implement logic to feed the right entity class into the engine via user input for Hibernate Search GenericJPA. Entity classes have to be marked with **@InIndex** on the type level so we can start from any object's class and then go up in the class hierarchy until we find one that is annotated with this annotation. If no **@InIndex** is found, we use the actual class of the entity object we are about to index as a best effort (this is the behaviour Hibernate Search ORM has). This algorithm is described in Java code in the next listing 27:

```

1 //get the first class in the hierarchy
2 Class<T> clazz = (Class<T>) entity.getClass();
3
4 //check if the original class has @InIndex present
5 //if yes, we don't have to go higher up in the class hierarchy
6 if ( !clazz.isAnnotationPresent( InIndex.class ) ) {
7
8     //go up in the class hierarchy until either a @InIndex is found
9     //or there is no superclass anymore.
10    while ( (clazz = (Class<T>) clazz.getSuperclass()) != null ) {
11        if ( clazz.isAnnotationPresent( InIndex.class ) ) {
12            break;
13        }
14    }
15 }
16
17
18 //if we have found a class annotated with @InIndex
19 //we return it here
20 if ( clazz != null ) {
21     return clazz;
22 }
23
24 //no @InIndex found, try the entities direct class
25 //as a best effort
26 return entity.getClass();

```

Listing 27: Algorithm to determine the actual indexed type

Note that every entity that is part of the index has to be annotated with `@InIndex`, even the ones that are just embedded. With this in mind our entities `Book` and `Author` now look like this:

```
1 @Entity
2 @InIndex
3 @Table(name = "Book")
4 @Indexed
5 public class Book {
6
7     //rest is unchanged
8
9 }
```

Listing 28: Book.java with `@InIndex`

```
1 @Entity
2 @InIndex
3 @Table(name = "Author")
4 public class Author {
5
6     //rest is unchanged
7
8 }
```

Listing 29: Author.java with `@InIndex`

A similar behaviour supporting the subclassing of entities can be achieved with JPA's `@Entity` replacing the `@InIndex` annotation as these annotations can be found on the first real entity class in the hierarchy as well. We didn't choose this approach because by using `@InIndex` we support indexing of non-JPA entities as well. A hybrid approach checking for both annotations might be possible, but using only `@InIndex` is sufficient for now.

6.3.3 Queries

While we didn't mention this in chapter 6.2.2, Hibernate Search ORM supports modifying the resulting objects of a query with these two methods on **FullTextQuery**:

- **setCriteriaQuery(Criteria criteria)**: This method lets the user define a custom Hibernate Criteria query (no JPA criteria query) that has to be used to retrieve the results from the database. This can be used to make sure all necessary data is loaded after it is returned by `getResultList()`. These custom queries are used in cases where no session is available anymore when the data is finally used: If the data is requested, an error would occur.
- **setResultTransformer(ResultTransformer resultTransformer)**: A ResultTransformer can be used to transform the results (useful for projections) into POJOs (Plain Old Java Object).

There is a problem with these two methods, though. They are using the Hibernate ORM API to accomplish their behaviour, and therefore we cannot support the methods on our generic version of the interface.

By adding a new method **entityProvider(EntityProvider entityProvider)** with the same EntityProvider interface as in chapter 5.3.3 to the method, we can at least support custom queries.

As the main use case scenario for the ResultTransformer is probably just the transformation from a projection of the queried documents to a POJO, we just completely remove this feature. In the future, we can add such a feature back to the generic version, if needed. But as this method cannot be kept as-is anyways, Hibernate Search ORM developers wanting to use Hibernate Search GenericJPA that use this feature have to change some of their code either way.

Besides these changes, the interface behaves the exact same as described in 6.2.2.

6.3.4 Index rebuilds

The `MassIndexer` utility is a really important feature of Hibernate Search ORM. As it uses Hibernate ORM logic under the hood (and in its interface), we have to write our own version of it. We don't build an API compatible version for Hibernate Search GenericJPA as a `MassIndexer` is generally not used in many places in the code anyways. Additionally this way we can give different configuration properties for better performance as our implementation differs in some details.

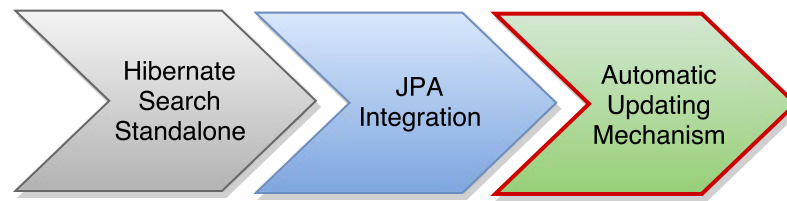
The basic ideas are the same though: Each entity type has its ids scrolled from the database by one thread (there can be multiple threads doing this, but for other entities). Then, a configurable amount of indexing threads handles these ids batch by batch in a Hibernate Search index-writing backend optimized for this task (this is part of Hibernate Search's engine is therefore reused).

In Hibernate Search GenericJPA our `Book` entities are massindexed like this:

```
1 EntityManager em = ...;
2 FullTextEntityManager fem = Search.getFullTextEntityManager(em);
3
4 fem.createIndexer( Book.class )
5     .batchSizeToLoadObjects( 25 )
6     .threadsToLoadObjects( 12 )
7     .batchSizeToLoadIds( 150 )
8     .idProducerTransactionTimeout( 1800 )
9     .startAndWait();
```

Listing 30: `MassIndexer` usage with Hibernate Search ORM

7 Automatic index updating



As already stated in chapter 4.4, the automatic index updating feature is required for a reasonable Hibernate Search GenericJPA. As this is arguably the most complicated feature for GenericJPA, we will go into detail about how we are achieving it next. We will start by giving a description of the different implementations available and then decide which ones to use. We are however not showing the complete internal code architecture - like in chapters 5 and 6 - in favour of explaining in detail how the general ideas work. After that, we will also give a short overview of the pros and cons of the chosen approaches.

7.1 Description of different implementations

There are several approaches to building an automatic index updating feature. While they are all different in the specifics, they can generally be separated into two categories: **synchronous** and **asynchronous**. Synchronous in this context means that the index is updated as soon as the newly changed data is persisted in the database without any real delay while in an asynchronous updating mechanism an arbitrary amount of time passes before the index is updated. While synchronous approaches are needed in some rare cases, fulltext search generally doesn't require a 100% up-to-date index at every point in time as a search index generally is not the source of truth in an application (only the database contains the "truth").

We will now work out a solution for both the synchronous and asynchronous case, while the asynchronous version will serve as a backup whenever the synchronous mechanism is not applicable.

7.1.1 Synchronous approach

For the synchronous approach we have two candidates: A system based on JPA callback events and another one that uses the native APIs of JPA providers. We start with the JPA callbacks and then go onto the native APIs.

7.1.1.1 JPA events As we are trying to work with as little vendor specific APIs, JPA's callback events look like a suitable candidate for listening to changes in entities.

To listen for the JPA events we have two options: annotate the entities with callback methods or create a separate listener class. We will only take a look at the listener class since we don't want to have unnecessary methods in a possible user's entities. This listener class doesn't have to implement an interface, but must have methods annotated with special annotations. The relevant ones are `@PostPersist`, `@PostUpdate`, `@PostDelete` (there are "pre-versions" available as well, but we focus on the post methods as they are more useful). What each specific annotation stands for is quite self-explanatory.

A listener class generally looks like this:

```
1 public class EntityListener {  
2  
3     @PostPersist  
4     public void persist(Object entity) {  
5         //handle the event  
6     }  
7  
8     @PostUpdate  
9     public void update(Object entity) {  
10        //handle the event  
11    }  
12  
13    @PostDelete  
14    public void delete(Object entity) {  
15        //handle the event  
16    }  
17  
18 }
```

Listing 31: Example JPA entity listener

These EntityListeners are generally applied with an annotation on the entity:

```
1 @EntityListeners( { EntityListener.class } )
2 public class Book {
3
4     //...
5
6 }
```

Listing 32: Using a JPA entity listener

As the JPA provider creates the EntityListeners automatically, we have no access to them without injecting a reference to them in a static way. While this might cause some Classloader problems, it should be fine in most cases.

```
1 public class EntityListener {
2
3     public EntityListener() {
4         // inject it somewhere
5         // so we can access it in a static way
6         EntityListenerRegistry.inject(this);
7     }
8
9     //...
10
11 }
```

Listing 33: Injecting the EntityListener

Even though these listeners seem to be the perfect fit as they would enable us to fully integrate only with JPA interfaces, they have two big issues as we find out after investigating further.

Firstly, not all JPA providers seem to handle these events similarly: For example Hibernate ORM doesn't propagate events from collection tables to the owning entity, while EclipseLink does (EclipseLink's behaviour would be needed from all providers).

Secondly, we find out that the events are triggered on flush instead of commit as can be seen in listing 34. This is an issue if the changed data is not actually committed:

```
1 EntityManager em = ...;
2
3 em.getTransaction().begin();
4
5 Book book = em.find( Book.class "someIsbn" );
6 book.setTitle( "someNewTitle" );
7
8 // flushes , so we retrieve the Book with the changes from above
9 // => event is triggered
10 List<Book> allBooks =
11     em.createQuery( "SELECT b FROM Book b" ).getResultList();
12
13 // we have no way to get this event to revert the wrong index change
14 em.getTransaction().rollback();
```

Listing 34: Event triggering on flush

While it **might** be possible to somehow fix the flush issue, the general bad support from JPA providers like Hibernate ORM renders this approach unusable until the JPA providers work the same way regarding the event propagation to some reasonable extent.

7.1.1.2 Native integration with JPA providers Almost every JPA provider has its own internal event system that is useful for cache invalidation and other tasks. These combined with hooks into the transaction management allow us to build a proper index updating system that works with transactions in mind (big improvement compared to the `flush()` issues of plain JPA).

JPA providers generally have callbacks similar to those of the JPA events (no knowledge about database specifics is needed, Java types are used), but also provide additional information about the database session that caused the changes.

By definition, these kind of integrations are not portable between JPA providers and require us to write different systems for all the JPA providers. But as the landscape for popular JPA providers probably only consists of Hibernate ORM, EclipseLink and OpenJPA, we can implement listeners for these and the others will have to rely on the asynchronous backup approach (as of the time of writing this, we have only implemented integrations for Hibernate ORM and EclipseLink).

As this seems to be the only reasonable solution for a synchronous update system, we are using it for Hibernate Search GenericJPA even though it is no real native solution because of the JPA implementation dependent code.

Note: we don't describe how these event systems are built in particular as they differ a lot in their APIs, but generally these are straightforward to use and describing the implementations would be unspectacular.

7.1.2 Asynchronous approach

In contrary to the synchronous approach where we described two different versions, for the asynchronous version we only have one feasible solution available: a trigger based system.

Paul DuBois writes in MySQL - Developer's Library:

A Trigger is a stored program that is associated with a particular table and is defined to activate for INSERT, DELETE or UPDATE statements for that table. A trigger can be set to activate either before or after each row processed by the statement. The trigger definition includes a statement that executes when the trigger activates.

[...]

A trigger can examine the current contents of a row before it is deleted or updated. This capability can be exploited to perform logging of changes [...].⁵²

While the quote above is meant to be for MySQL databases, many other RDBMSs support at least triggers on the three crucial events for event-listening: INSERT (CREATE), UPDATE, DELETE, just like MySQL^{53 54 55}.

In order to have triggers being useful for updating our Hibernate Search index, we have to get info about the events from the database back into our Java application. Since we cannot necessarily call Java code from our database (with the exception of some enterprise and in-memory databases), we have to write data about changes into auxiliary tables and then poll these regularly.

One benefit of this approach is that by using polling from the tables and the fact that triggers are executed in the same transaction as the original changing query, we don't have to manually hook into transactions or deal with data that has not been committed, yet, in general. If we do things right, we can even improve indexing performance by this: We can query for the latest event for each entity only, so we don't use up an unnecessary amount of CPU-time, but still keep the index up-to-date.

⁵²MySQL - Developer's Library, see [41]

⁵³CREATE TRIGGER in PostgreSQL, see [42]

⁵⁴Triggers in HSQLDB, see [43]

⁵⁵Triggers in Firebird, see [44]

7.1.2.1 Trigger architecture Triggers are generally created on tables. Since we want to use them for event-listening, we have to cover every table of the domain model that contains data indexed/stored in the index. This also includes all of the mapping tables between entities and all other secondary tables.

The following figure 13 shows the trigger architecture needed for our Author and Book example. Also note that we are using triggers that execute before changes are persisted.

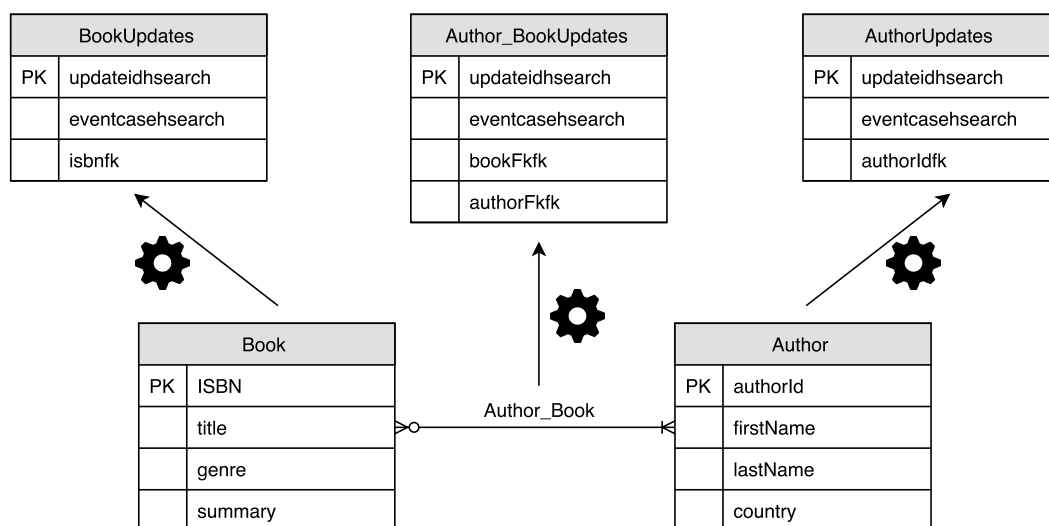


Figure 13: Triggers for the example project

All three tables Author, Book and Author_Book have three triggers registered on them (one for each event type). These triggers then fill up the update tables AuthorUpdates, BookUpdates and Author_BookUpdates (these names are just for demonstrative purposes) with info about occurring events. We can see that these update tables host at least three things:

1. **updateid primary key:** Update events have to be sortable by the order they occurred. All update tables share the same sequence of primary keys so that no key appears twice in all of these tables.
2. **eventcase column:** This column contains a identifier for the cases INSERT, DELETE or UPDATE.
3. **pseudo foreign key(s):** The relevant primary keys of the entities involved in the tables have to be stored in the Update tables as well. Note that they are not marked as real foreign keys as a DELETE event wouldn't work as we can't have a reference to a non existent entity.

7.1.2.2 Table creation Since the creation of these tables requires a lot of work to be done, we have to automate it as well as possible. We do this by requiring additional **@UpdateInfo** annotations on the entities to map the required information for the update tables and then generating them out of it.

These annotations contain at least the original table's name (`UpdateInfo#tableName`) and the names & types (`IdColumn#column` & `IdColumn#columnType`) of the entity key columns. The name of the update table and the columns in it are then generally derived automatically from that.

A similar behaviour might be possible by using the JPA mapping annotations to read the original schema and then deduce the needed update schema from that. We don't use this approach nonetheless, because the task of parsing these annotations correctly would be prone to errors due to the amount of different annotations (`@Basic`, `@Column`, `@IdClass`, `@EmbeddedCollection`, `@OneToOne`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`, ...). In some cases these annotations aren't even required for JPA to work, which makes it even more complicated. This makes the approach less streamlined than using the extra `@UpdateInfo` annotations.

The following listings show the `@UpdateInfo` annotation in use:

```
1 @Entity
2 @InIndex
3 @Table(name = "Book")
4 @Indexed
5 @UpdateInfo(
6     tableName = "Book",
7     idInfos = @IdInfo(
8         columns = @IdColumn(
9             column = "isbn",
10            columnType = ColumnType.STRING
11        )
12    )
13 )
14 public class Book {
15
16     // ... unchanged.
17
18     // mapping table events handled on Author side
19
20     // getters & setters ...
21 }
```

Listing 35: Book.java with Hibernate Search annotations

```
1 @Entity
2 @InIndex
3 @Table(name = "Author")
4 @UpdateInfo(
5     tableName = "Author",
6     idInfos = @IdInfo(
7         columns = @IdColumn(
8             column = "authorId",
9             columnType = ColumnType.LONG
10        )
11    )
12 )
13 public class Author {
14
15     // ... unchanged.
16
17     @UpdateInfo(tableName = "Author_Book",
18         idInfos = {
19             @IdInfo(entity = Author.class,
20                 columns = @IdColumn(
21                     column = "authorFk",
22                     columnType = ColumnType.LONG
23                 )
24             ),
25             @IdInfo(entity = Book.class,
26                 columns = @IdColumn(
27                     column = "bookFk",
28                     columnType = ColumnType.STRING
29                 )
30             )
31         })
32     private Set<Book> books;
33
34     //getters & setters ...
35 }
```

Listing 36: Author.java with Hibernate Search annotations

Note: The update tables are NOT JPA entities, so we have to work with native SQL in the backend

If the developer needs different names for the update tables and their columns (e.g. if there already exists a table with the same name), it is possible to manually set these. They can be found on the same level in the annotations as the corresponding info for the original table is set.

Options for multivalued keys and custom column types are also available as by default only singular valued keys of the column types corresponding to Java's Integer, Long and String are supported. While we don't go into detail how these expert features are used, information about how to use them can be found in the JavaDoc of the annotations.

Since database triggers and tables are not created the same on every RDBMS, we build an abstraction to get the necessary SQL code. This is done with the **TriggerSQLStringSource** interface. Its implementations return the specific SQL strings working on the corresponding RDBMS. As of this writing we have implementations for MySQL, PostgreSQL and HSQLDB. See the property "hibernate.search.trigger.source" in table 2 in the appendix for information about how to set the correct one for each database.

This table also contains a property called "hibernate.search.trigger.createstrategy" that controls whether and how the triggers and tables are generated at all. If automatic trigger creation is disabled, the user still has to provide the information about the update tables that should be used for updating with the annotations as described above.

7.1.2.3 Event retrieval Now that we know how the events are stored in the update tables, we will describe an efficient way to query the database for these entries.

We only need the latest event for each entity (or combination of entities for mapping tables). The following SQL query shown in listing 37 is doing this for the table `author_bookupdates` with standard SQL that should be working on every RDBMS:

```

1 SELECT t1.updateidhsearch , t1.authorFkfk , t1.bookFkfk
2 FROM author_bookupdates t1
3 INNER JOIN
4 (
5     /* select the most recent update */
6     SELECT max(t2.updateidhsearch) updateid ,
7             t2.authorFkfk , t2.bookFkfk
8     FROM author_bookupdates t2
9     GROUP BY t2.authorFkfk , t2.bookFkfk
10 ) t3 on t1.updateidhsearch = t3.updateid
11 /* handle events that occurred earlier first */
12 ORDER BY t1.updateidhsearch ASC;
```

Listing 37: Querying for updates (Author_Book)

We run queries of this type for every update table with fixed delays (configurable, see property "hibernate.search.trigger.updateDelay" in table 2). Then, we scroll from the results of these queries simultaneously while ordering by the updateids between the queries to make sure the events are definitely handled in the right order (see listing 47 in the appendix).

This information is all we need to keep our index up-to-date. For the INSERT and UPDATE case we can just query the database for a new version and pass that to the engine. For the DELETE case we have to work directly on the index and also have to enforce `@IndexedEmbedded#includeEmbeddedObjectId = true` on the entities. This is required so that we can determine the root entity in the index as its entry has to be updated additionally if the original entity is changed (An entity contained in one index can have its own index as well).

After the index is updated accordingly, we run a delete query that deletes all update events having an updateid lower than the last processed one for each table.

We don't use a TRUNCATE statement for the query shown in the following listing 38 as it was only introduced with the SQL:2008 standard ⁵⁶, which some RDBMSs don't fully support ⁵⁷. Using TRUNCATE could therefore be a deal-breaker for some people wanting to use Hibernate Search GenericJPA. With the DELETE FROM query we make sure the clean-up statement is supported by as many RDBMSs as possible (older versions included).

```
1 DELETE FROM author_bookupdates WHERE updateidhsearch < #last_handled_id#
```

Listing 38: Deleting handled updates (Author_Book)

With the two queries described in this section we are able to keep the index up-to-date efficiently and also make sure that no event is handled twice.

⁵⁶Truncate statement PostgreSQL docs, see [45]

⁵⁷Firebird conformance, see [46]

7.2 Comparison of approaches

We already discussed the differences of synchronous and asynchronous approaches in general earlier this chapter. Additionally to that, the two chosen implementations differ in terms of extra work that has to be done to get them to work (user-friendliness for the developer) and features.

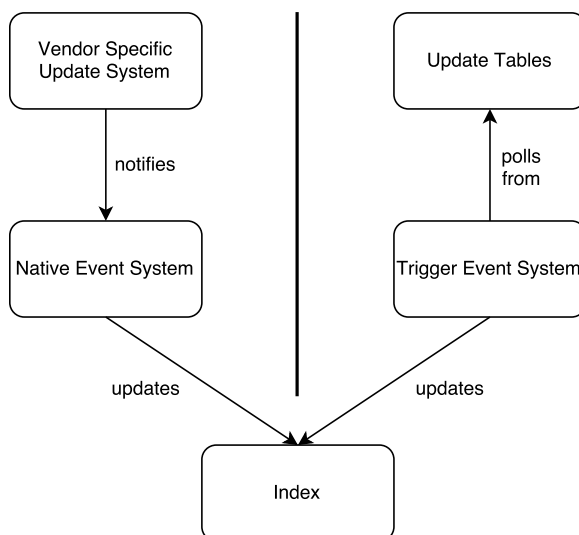


Figure 14: Hibernate Search GenericJPA update mechanisms

7.2.1 Additional work

Since the native event system gets the proper information about changes from the vendor side, it doesn't require a lot of information about the general structure of the domain model and tables in the database. The Trigger based system however does need extra information as it has to poll info about changes from the database. This is the reason the user has to add this information as we have seen in chapter 7.1.2.2.

7.2.2 Features

The native event system has the exact same updating behaviour as Hibernate Search ORM's update mechanism because it works on the same principles of using the existing event APIs. It just works for more ORM providers.

With this similarity come two important drawbacks:

1. It (the mechanism) only works with specifically supported JPA APIs
2. Database changes coming from anything else than JPA APIs are not recognized and includes native SQL queries from EntityManagers. This also means that the database can only be used by the JPA application and no other programs should have write access to the database.

These two drawbacks are non-existent with the trigger event system as it doesn't require any specific JPA implementation (1) and works on the database level (2).

7.2.3 Summary

The following table 1 summarizes all pros and cons - including the ones for being synchronous or asynchronous - once again:

Approach	Pros	Cons
Native Event System	<ul style="list-style-type: none"> + no additional work needed by the developer + 100 % up-to-date index all the time 	<ul style="list-style-type: none"> - relies on different implementation-specific APIs (only works with specifically supported ones) - changes from outside of the JPA provider are not recognized (e.g. native SQL access)
Trigger Event System	<ul style="list-style-type: none"> + works with any JPA implementation (even rarely used ones) + changes from outside of the JPA provider are recognized (e.g. native SQL access) 	<ul style="list-style-type: none"> - additional work by the developer needed (annotations) - unsuitable in cases that need a 100% up-to-date index all the time

Table 1: Pros and Cons of the two update systems

8 Usage of Hibernate Search GenericJPA

Having described how Hibernate Search GenericJPA works and how it is designed we will now take a look at how it can be used in our example project from chapter 4.1. While having already explained this part by part in each chapter, the following is everything put together. For updating, we use the asynchronous updating mechanism as described in chapter 7.1.2.

8.1 Dependencies

The following example needs to have at least these dependencies on the classpath:

1. EclipseLink 2.5.0
2. HSQLDB 2.3.3 (in memory database)
3. Hibernate Search GenericJPA

8.2 Entities

First, we have to update the Entity mappings in the Java classes. We add the **@Indexed**, **@DocumentId**, **@Field**, **@IndexedEmbedded**, **@ContainedIn** as already known from the original Hibernate Search ORM (chapter 5.1). Using Hibernate Search GenericJPA then requires us to add the **@InIndex** on every entity contained in the index as described in chapter 6.3.2. Because we are using the asynchronous updating mechanism here, we have to add information about how to create the update tables as well (chapter 7.1.2.2).

The resulting entities with the changes highlighted look like this:

```
1 @Entity
2 @Table(name = "Book")
3 @InIndex
4 @Indexed
5 @UpdateInfo(tableName = "Book",
6             idInfos = @IdInfo(
7                 columns = @IdColumn(
8                     column = "isbn",
9                     columnType = ColumnType.STRING))
10 public class Book {
11
12     @Id
13     @DocumentId
14     @Column(name = "isbn")
15     private String isbn;
16
17     @Column(name = "title")
18     @Field
19     private String title;
20
21     @Column(name = "genre")
22     @Field
23     private String genre;
24
25     @Lob
26     @Column(name = "summary")
27     @Field
28     private String summary;
29
30     @ManyToMany(mappedBy = "books", cascade = {
31         CascadeType.MERGE,
32         CascadeType.DETACH,
33         CascadeType.PERSIST,
34         CascadeType.REFRESH
```

```

35     })
36     @IndexedEmbedded(includeEmbeddedObjectId = true)
37     private Set<Author> authors;
38
39     // getters & setters ...
40
41 }

```

Listing 39: Complete Book.java

```

1  @Entity
2  @Table(name = "Author")
3  @InIndex
4  @UpdateInfo(tableName = "Author",
5      idInfos = @IdInfo(
6          columns = @IdColumn(
7              column = "authorId",
8              columnType = ColumnType.LONG
9          )
10 ))
11 public class Author {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.AUTO)
15     @Column(name = "authorId")
16     @DocumentId
17     private Long authorId;
18
19     @Column(name = "firstName")
20     @Field
21     private String firstName;
22
23     @Column(name = "lastName")
24     @Field
25     private String lastName;
26
27     @Column(name = "country")
28     @Field
29     private String country;
30
31     @ManyToMany(cascade = {
32         CascadeType.MERGE,
33         CascadeType.DETACH,
34         CascadeType.PERSIST,
35         CascadeType.REFRESH
36     })
37     @JoinTable(name = "Author_Book",
38         joinColumns = @JoinColumn(name = "authorFk",
39             referencedColumnName = "authorId"),

```

```
40         inverseJoinColumns = @JoinColumn(name = "bookFk",
41                                           referencedColumnName = "isbn"))
42     @UpdateInfo(tableName = "Author_Book",
43               idInfos = {
44                 @IdInfo(entity = Author.class,
45                       columns = @IdColumn(
46                         column = "authorFk",
47                         columnType = ColumnType.LONG) ),
48                 @IdInfo(entity = Book.class,
49                       columns = @IdColumn(
50                         column = "bookFk",
51                         columnType = ColumnType.STRING) )
52     })
53     @ContainedIn
54     private Set<Book> books;
55
56     // getters & setters ...
57
58 }
```

Listing 40: Complete Author.java

8.3 persistence.xml

The persistence.xml file for our JPA based project is straightforward. As we are using an in-memory database with HSQLDB, settings for the schema creation and the user management are not important as the database is recreated at every restart.

```

1 <persistence xmlns="http://java.sun.com/xml/ns/persistence"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
4     http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
5     version="2.0">
6
7   <persistence-unit name="EclipseLink_HSQLDB"
8     transaction-type="RESOURCE_LOCAL">
9     <provider>
10       org.eclipse.persistence.jpa.PersistenceProvider
11     </provider>
12     <class>*.*.Author</class>
13     <class>*.*.Book</class>
14     <properties>
15       <property name="javax.persistence.jdbc.driver"
16         value="org.hsqldb.jdbcDriver"/>
17       <property name="javax.persistence.jdbc.url"
18         value="jdbc:hsqldb:mem:test"/>
19       <property name="javax.persistence.jdbc.user"
20         value="user"/>
21       <property name="javax.persistence.jdbc.password"
22         value="password"/>
23       <property name="eclipselink.ddl-generation"
24         value="drop-and-create-tables"/>
25       <property name="eclipselink.logging.level"
26         value="INFO"/>
27       <property name="eclipselink.ddl-
28         generation.output-mode"
29         value="both"/>
30     </properties>
31   </persistence-unit>
32
33 </persistence>

```

Listing 41: Complete persistence.xml

8.4 Code usage example

In the following listing we show the whole lifecycle of a Hibernate Search GenericJPA based application. The relevant code passages are commented in the code.

```
1 Properties properties = new Properties();
2
3 // use the async backend
4 properties.setProperty(
5     "hibernate.search.searchfactory.type",
6     "sql"
7 );
8
9 // we are using HSQLDB, so use the right TriggerSource
10 properties.setProperty(
11     "hibernate.search.trigger.source",
12     "org.hibernate.search.genericjpa.db." +
13         "events.triggers.HSQLDBTriggerSQLStringSource"
14 );
15
16 // start up the EntityManagerFactory (entry-point to JPA)
17 // and create one EntityManager
18 EntityManagerFactory emf = Persistence
19     .createEntityManagerFactory( "EclipseLink_HSQLDB" );
20 EntityManager em = emf.createEntityManager();
21
22 // start up Hibernate Search GenericJPA
23 JPASearchFactoryController searchController =
24     Setup.createSearchFactoryController( emf, properties );
25
26 // persist entities in the database
27 em.getTransaction().begin();
28 Author author = ...;
29 Book book = ...;
30 book.setAuthor( author );
31 em.persist( em );
32 em.getTransaction().commit();
33
34 // we are using an async backend, so wait a bit
35 // for the updating mechanism to handle the
36 // persist (Exception not handled here)
37 Thread.sleep( 10_000 );
38
39 // create a FullTextEntityManager
40 FullTextEntityManager fem = searchController
41     .getFullTextEntityManager( em );
42
```

```
43 // query for all Books having the title "searchString"
44 FullTextQuery fullTextQuery = fem.createFullTextQuery(
45     fem.getSearchFactory().buildQueryBuilder()
46         .forEntity( Book.class )
47         .get()
48         .keyword()
49         .onField( "title" )
50         .matching( "searchString" )
51         .createQuery() ,
52     Book.class );
53
54 List<Book> books = (List<Book>) fullTextQuery.getResultList();
55
56 //handle the books
57 System.out.println( books );
58
59 // close everything
60 // (FullTextEntityManager is not closed because
61 // the EntityManager is closed)
62 em.close();
63 searchController.close();
64 emf.close();
```

Listing 42: Complete usage

Note that we didn't put the code into a main method. This is due to the fact that in a real application all this code would obviously not be put into one single method.

The startup process of Hibernate Search GenericJPA is generally put into an extra lifecycle helper that stores a reference to the JPASearchFactoryController in a global variable upon application startup similar to what is generally done with JPA's EntityManagerFactory (at least in Java SE applications). All Search related code then acquires the reference to the JPASearchFactoryController from the global variable and uses it similarly to the above code. The lifecycle helper is also responsible for closing the JPASearchFactoryController when the application is shutting down.

9 Outlook

In this thesis we described how we can integrate Hibernate Search with JPA conform ORM implementations. We started by building a standalone integration of hibernate-search-engine, then integrated it with JPA and finally created an automatic index updating mechanism. All challenges described in chapter 4 have been resolved.

The only feature needing some extra work is probably the generic updating mechanism with database triggers. At the moment the developer has to specify additional annotations containing information about the update tables by hand. As mentioned in chapter 7.1.2.2, at least some of the information is known to be able to be retrieved directly from JPA annotations. These mechanisms are not included in this thesis but can be added in a future version.

During the process of designing and writing the code for Hibernate Search GenericJPA we tried to be as compatible with the original Hibernate Search API as possible. While one reason for this is to make the switch easier for developers that want to try it out, the biggest one is that the ultimate goal for this project is to be merged into the original Hibernate Search codebase even though we haven't mentioned this in the beginning.

This is also why this project has to be looked as a proof of concept regardless of the fact that the code as it can be found on GitHub ⁵⁸ can already be used in real applications. To make sure of this, every relevant part of Hibernate Search GenericJPA has been extensively tested in single feature-tests and integration-tests as described in chapter 2. Hibernate Search GenericJPA can therefore be considered stable.

The first steps of the merging process have already been discussed with the Hibernate Search development team and work on it is to be started in November 2015. This comes exactly at the right moment as the Hibernate Search team is planning API changes in the near future ⁵⁹ and some interfaces have to be altered (as seen in chapter 6) in order to support generic JPA.

As soon as the generic version is part of Hibernate Search and is fully compatible with its API, Hibernate Search could be looked at as a de-facto standard for fulltext search in JPA. Having such a standard would be quite beneficial for the JPA world as smaller JPA providers could have a better chance at getting a bigger user base, which is good for research and innovation.

⁵⁸Hibernate Search GenericJPA GitHub repository, see [47]

⁵⁹Hibernate Search roadmap, see [38]

Used software

For the development of Hibernate Search GenericJPA as described in this thesis we have used the following software and libraries (only the most relevant listed here, for more information check the pom.xml files in the GitHub repository ⁶⁰):

Hibernate Search related libraries:

- Hibernate Search 5.5.0.Alpha1 (especially hibernate-search-engine)
- Lucene 5.2.1 (included in Hibernate Search)
- Infinispan Directory Provider 8.0.0.Beta3

Databases:

- HSQLDB 2.3.3
- MySQL Community Edition 5.5
- MariaDB 10.0.17
- PostgreSQL 9.4.4

JPA providers:

- EclipseLink 2.5.0
- Hibernate ORM 4.3.9
- OpenJPA 2.4.0

Application servers:

- Glassfish Embedded 4.1
- Wildfly 8.2.0.Final
- TomEE 1.7.2

Building tools:

- JUnit 4.11
- Arquillian 1.1.8.Final
- Maven 3.3.1

⁶⁰Hibernate Search GenericJPA GitHub repository, see [47]

Listings

Following are some interesting classes referenced in the thesis that were too long to fit into the text.

Transaction:

This class is the simple Transaction representation used to control index changes. It is not intended to be similar to a RDBMS transaction, but is merely a batch context with simple commit and rollback features.

```
1 public class Transaction implements TransactionContext {
2
3     private boolean progress = true;
4     private List<Synchronization> syncs = new ArrayList<>();
5
6     @Override
7     public boolean isTransactionInProgress() {
8         return this.progress;
9     }
10
11    @Override
12    public Object getTransactionIdentifier() {
13        return this;
14    }
15
16    @Override
17    public void registerSynchronization(
18        Synchronization synchronization ) {
19        this.syncs.add( synchronization );
20    }
21
22    /**
23     * @throws IllegalStateException if already committed/rolledback
24     */
25    public void commit() {
26        if ( !this.progress ) {
27            throw new IllegalStateException(
28                "can't commit - " +
29                "No Search Transaction is in Progress!" );
30        }
31        this.progress = false;
32        this.syncs.forEach( Synchronization::beforeCompletion );
33
34        for ( Synchronization sync : this.syncs ) {
35            sync.afterCompletion( Status.STATUS_COMMITTED );
36        }
37    }
38 }
```



```
37     }
38
39     /**
40      * @throws IllegalStateException if already committed/rolledback
41      */
42     public void rollback() {
43         if ( !this.progress ) {
44             throw new IllegalStateException(
45                 "can't rollback - " +
46                 "No Search Transaction is in Progress!" );
47         }
48         this.progress = false;
49         this.syncs.forEach( Synchronization::beforeCompletion );
50
51         for ( Synchronization sync : this.syncs ) {
52             sync.afterCompletion( Status.STATUS_ROLLEDBACK );
53         }
54     }
55
56 }
```

Listing 43: the simple Transaction contract

StandaloneSearchConfiguration:

hibernate-search-engine requires an object implementing the SearchConfiguration interface. StandaloneSearchConfiguration is the basic implementation of this used in our standalone version of Hibernate Search.

```
1  /**
2   * Manually defines the configuration.
3   * Classes and properties are the only implemented options at the moment.
4   *
5   * @author Martin Braun (adaption), Emmanuel Bernard
6   */
7  public class StandaloneSearchConfiguration
8      extends SearchConfigurationBase
9      implements SearchConfiguration {
10
11      private final Logger LOGGER =
12          Logger.getLogger(
13              StandaloneSearchConfiguration.class.getName()
14          );
15
16      private final Map<String, Class<?>> classes;
17      private final Properties properties;
18      private final HashMap<Class<? extends Service>, Object>
19          providedServices;
20      private final InstanceInitializer initializer;
21      private SearchMapping programmaticMapping;
22      private boolean transactionsExpected = true;
23      private boolean indexMetadataComplete = true;
24      private boolean idProvidedImplicit = false;
25      private ClassLoaderService classLoaderService;
26      private ReflectionManager reflectionManager;
27
28      public StandaloneSearchConfiguration() {
29          this( new Properties() );
30      }
31
32      public StandaloneSearchConfiguration(Properties properties) {
33          this(
34              SubClassSupportInstanceInitializer.INSTANCE,
35              properties
36          );
37      }
38
39      public StandaloneSearchConfiguration(InstanceInitializer init) {
40          this( new Properties() );
41      }
42  }
```

```
43     public StandaloneSearchConfiguration(InstanceInitializer init ,
44         Properties properties) {
45         this.initializer = init;
46         this.classes = new HashMap<>();
47         this.properties = properties;
48         // default values if nothing was explicitly set
49         this.properties.computeIfAbsent(
50             "hibernate.search.default.directory_provider",
51             (key) -> {
52                 LOGGER.info(
53                     "defaulting to RAM directory-provider"
54                 );
55                 return "ram";
56             });
57         this.properties.computeIfAbsent(
58             "hibernate.search.lucene_version",
59             (key) -> {
60                 LOGGER.info(
61                     "defaulting to Lucene Version: "
62                     + Version.LUCENE_5_2_1.toString()
63                 );
64                 return Version.LUCENE_5_2_1.toString();
65             });
66         this.reflectionManager = new JavaReflectionManager();
67         this.providedServices = new HashMap<>();
68         this.classLoaderService = new DefaultClassLoaderService()
69         ;
70     }
71
72     public StandaloneSearchConfiguration addProperty(String key,
73         String value) {
74         properties.setProperty( key, value );
75         return this;
76     }
77
78     public StandaloneSearchConfiguration addClass(Class<?> indexed) {
79         classes.put( indexed.getName(), indexed );
80         return this;
81     }
82
83     @Override
84     public Iterator<Class<?>> getClassMappings() {
85         return classes.values().iterator();
86     }
87
88     @Override
89     public Class<?> getClassMapping(String name) {
90         return classes.get( name );
91     }
```

```
90     }
91
92     @Override
93     public String getProperty(String propertyName) {
94         return properties.getProperty( propertyName );
95     }
96
97     @Override
98     public Properties getProperties() {
99         return properties;
100     }
101
102     @Override
103     public ReflectionManager getReflectionManager() {
104         return this.reflectionManager;
105     }
106
107     @Override
108     public SearchMapping getProgrammaticMapping() {
109         return programmaticMapping;
110     }
111
112     public StandaloneSearchConfiguration setProgrammaticMapping(
113         SearchMapping programmaticMapping
114     ) {
115         this.programmaticMapping = programmaticMapping;
116         return this;
117     }
118
119     @Override
120     public Map<Class<? extends Service>, Object>
121         getProvidedServices() {
122         return providedServices;
123     }
124
125     public void addProvidedService(
126         Class<? extends Service> serviceRole ,
127         Object service
128     ) {
129         providedServices.put( serviceRole , service );
130     }
131
132     @Override
133     public boolean isTransactionManagerExpected() {
134         return this.transactionsExpected;
135     }
136
137     public void setTransactionsExpected(
```

```
138         boolean transactionsExpected) {
139             this.transactionsExpected = transactionsExpected;
140         }
141
142         @Override
143         public InstanceInitializer getInstanceInitializer() {
144             return initializer;
145         }
146
147         @Override
148         public boolean isIndexMetadataComplete() {
149             return indexMetadataComplete;
150         }
151
152         public void setIndexMetadataComplete(
153             boolean indexMetadataComplete) {
154             this.indexMetadataComplete = indexMetadataComplete;
155         }
156
157         @Override
158         public boolean isIdProvidedImplicit() {
159             return idProvidedImplicit;
160         }
161
162         public StandaloneSearchConfiguration
163             setIdProvidedImplicit(boolean idProvidedImplicit) {
164             this.idProvidedImplicit = idProvidedImplicit;
165             return this;
166         }
167
168         @Override
169         public ClassLoaderService getClassLoaderService() {
170             return classLoaderService;
171         }
172
173         public void setClassLoaderService(
174             ClassLoaderService ) {
175             this.classLoaderService = classLoaderService;
176         }
177     }
178 }
```

Listing 44: StandaloneSearchConfiguration.java

BasicEntityProvider:

This is the basic implementation of the EntityProvider interface which is used to abstract the database access in the standalone version. It uses a JPA EntityManager to accomplish this.

```
1 public class BasicEntityProvider implements EntityProvider {
2
3     private static final String QUERY_FORMAT =
4         "SELECT obj FROM %s obj " +
5         "WHERE obj.%s IN :ids";
6     private final EntityManager em;
7     private final Map<Class<?>, String> idProperties;
8
9     public BasicEntityProvider(EntityManager em,
10        Map<Class<?>, String> idProperties) {
11         this.em = em;
12         this.idProperties = idProperties;
13     }
14
15     @Override
16     public void close() throws IOException {
17         this.em.close();
18     }
19
20     @Override
21     public Object get(Class<?> entityClass, Object id,
22        Map<String, String> hints) {
23         return this.em.find( entityClass, id );
24     }
25
26     @SuppressWarnings({"rawtypes", "unchecked"})
27     @Override
28     public List getBatch(Class<?> entityClass, List<Object> ids,
29        Map<String, String> hints) {
30         List<Object> ret = new ArrayList<>( ids.size() );
31         if ( ids.size() > 0 ) {
32             String idProperty =
33                 this.idProperties.get( entityClass );
34             String queryString =
35                 String.format(
36                     QUERY_FORMAT,
37                     this.em.getMetamodel()
38                         .entity( entityClass )
39                         .getName(),
40                     idProperty
41                 );
42             Query query = this.em.createQuery( queryString );
```

```
43         query.setParameter( "ids", ids );
44         ret.addAll( query.getResultList() );
45     }
46     return ret;
47 }
48
49 public void clearEm() {
50     this.em.clear();
51 }
52
53 public EntityManager getEm() {
54     return this.em;
55 }
56
57 }
```

Listing 45: BasicEntityProvider.java

Obtaining the idProperties:

This code snippet shows how the idProperties map needed for the instantiation of a BasicEntityProvider can be obtained. This mechanism is used on some other places of Hibernate Search GenericJPA as well.

```
1 SearchConfiguration config = ...;
2
3 MetadataProvider metadataProvider =
4     MetadataUtil.getDummyMetadataProvider( config );
5 MetadataRehasher rehasher = new MetadataRehasher();
6
7 List<RehashedTypeMetadata> rehashedTypeMetadatas = new ArrayList<>();
8 for ( Class<?> indexRootType : this.getIndexRootTypes() ) {
9     RehashedTypeMetadata rehashed =
10         rehasher.rehash(
11             metadataProvider
12                 .getTypeMetadataFor( indexRootType )
13         );
14     rehashedTypeMetadatas.add( rehashed );
15 }
16
17 Map<Class<?>, String> idProperties =
18     MetadataUtil.calculateIdProperties( rehashedTypeMetadatas );
```

Listing 46: Obtaining idProperties

MultiQueryAccess:

This is the utility class used to scroll results from multiple queries at once while retrieving the events from the database in the asynchronous approach.

```

1  /**
2   * Utility class that allows you to access multiple JPA queries at once.
3   * Data is retrieved from the database in batches
4   * and ordered by a given comparator.
5   * No need for messy Unions on the database level! <br>
6   * <br>
7   * This is particularly useful if you scroll all the data
8   * from the database incrementally and if you can
9   * compare in Code.
10  *
11  * @author Martin
12  */
13  public class MultiQueryAccess {
14
15      private final Map<String, Long> currentCountMap;
16      private final Map<String, Query> queryMap;
17      private final Comparator<ObjectIdentifierWrapper> comparator;
18      private final int batchSize;
19
20      private final Map<String, Long> currentPosition;
21      private final Map<String, LinkedList<Object>> values;
22
23      private Object scheduled;
24      private String identifier;
25
26
27      public MultiQueryAccess(
28          Map<String, Long> countMap,
29          Map<String, Query> queryMap,
30          Comparator<ObjectIdentifierWrapper> comparator,
31          int batchSize) {
32          if ( countMap.size() != queryMap.size() ) {
33              throw new IllegalArgumentException(
34                  "countMap.size() must be equal " +
35                      "to queryMap.size()" );
36          }
37          this.currentCountMap = countMap;
38          this.queryMap = queryMap;
39          this.comparator = comparator;
40          this.batchSize = batchSize;
41          this.currentPosition = new HashMap<>();
42          this.values = new HashMap<>();
43          for ( String ident : queryMap.keySet() ) {

```

```

44         this.values.put( ident , new LinkedList<>() );
45         this.currentPosition.put( ident , 0L );
46     }
47 }
48
49 private static int toInt(Long l) {
50     return (int) (long) l;
51 }
52
53 /**
54  * increments the value to be returned by {@link #get()}
55  *
56  * @return true if there is a value left to be visited
57  *       in the database
58  */
59 public boolean next() {
60
61     /*
62     *
63     *
64     * indentation broken to make this readable
65     *
66     *
67     */
68
69 this.scheduled = null;
70 this.identifier = null;
71 List<ObjectIdentifierWrapper> tmp =
72     new ArrayList<>( this.queryMap.size() );
73
74 for ( Map.Entry<String , Query> entry : this.queryMap.entrySet() ) {
75     String identifier = entry.getKey();
76     Query query = entry.getValue();
77     if ( !this.currentCountMap.get( identifier ).equals( 0L ) ) {
78         if ( this.values.get( identifier ).size() == 0 ) {
79             // the last batch is empty. get a new one
80             Long processed =
81                 this.currentPosition.get( identifier );
82             // yay JPA...
83             query.setFirstResult( toInt( processed ) );
84             query.setMaxResults( this.batchSize );
85             @SuppressWarnings("unchecked")
86             List<Object> list = query.getResultList();
87             this.values.get( identifier ).addAll( list );
88         }
89         Object val = this.values.get( identifier ).getFirst();
90         tmp.add( new ObjectIdentifierWrapper( val , identifier ) )
91
92     ;

```

```

91         }
92     }
93     tmp.sort( this.comparator );
94     if ( tmp.size() > 0 ) {
95         ObjectIdentifierWrapper arr = tmp.get( 0 );
96         this.scheduled = arr.object;
97         this.identifier = arr.identifier;
98         this.values.get( this.identifier ).pop();
99         Long currentPosition = this.currentPosition.get( arr.identifier )
100             ;
101         Long newCurrentPosition =
102             this.currentPosition
103                 .computeIfPresent( arr.identifier ,
104                     (clazz , old) -> old + 1 );
105         if ( Math.abs( newCurrentPosition - currentPosition ) != 1L ) {
106             throw new AssertionError(
107                 "the new currentPosition count " +
108                 "should be exactly 1 " +
109                 "greater than the old one" );
110         }
111         Long count = this.currentCountMap.get( arr.identifier );
112         Long newCount = this.currentCountMap.computeIfPresent(
113             arr.identifier , (clazz , old) -> old - 1
114         );
115         if ( Math.abs( count - newCount ) != 1L ) {
116             throw new AssertionError(
117                 "the new old remaining count " +
118                 "should be exactly 1 " +
119                 "greater than the new one" );
120         }
121     }
122     return this.scheduled != null;
123 }
124
125 /**
126  * @return the current value
127  */
128 public Object get() {
129     if ( this.scheduled == null ) {
130         throw new IllegalStateException(
131             "either empty or next() has " +
132             "not been called" );
133     }
134     return this.scheduled;
135 }
136
137 /**
138  * @return the identifier of the current value

```

```
138     */
139     public String identifier() {
140         if ( this.identifier == null ) {
141             throw new IllegalStateException(
142                 "either empty or next() has " +
143                 "not been called" );
144         }
145         return this.identifier;
146     }
147
148     public static class ObjectIdentifierWrapper {
149
150         public final Object object;
151         public final String identifier;
152
153         public ObjectIdentifierWrapper(Object object,
154             String identifier) {
155             this.object = object;
156             this.identifier = identifier;
157         }
158
159     }
160
161 }
```

Listing 47: MultiQueryAccess.java

Tables

This section contains all tables referenced in this thesis.

JPASearchFactoryController configuration:

When instantiating the JPASearchFactoryController with the Setup class the developer has to pass a property-Map (or a Java Properties) object. Besides containing the hibernate-search-engine configuration properties, some Hibernate Search GenericJPA configuration properties can be set in this map as well:

hibernate.search.useJTATransactions	false true
hibernate.search.searchfactory.type	sql manual-updates eclipselink hibernate openjpa
hibernate.search.trigger.batchSizeForUpdates	5
hibernate.search.trigger.batchSizeForUpdateQueries	20
hibernate.search.trigger.updateDelay	200
hibernate.search.trigger.source	<class>
hibernate.search.additionalIndexedTypes	<class>,<class>,...
hibernate.search.transactionManagerProvider	org.hibernate. search.generic jpa.trans action.impl JNDILookup Transaction ManagerProvider
hibernate.search.transactionManagerProvider.jndi	<jndi-string>
hibernate.search.trigger.createstrategy	create create-drop dont-create

Table 2: Basic JPASearchFactoryController configuration properties (**default**)

References

- [1] JSR 220: Enterprise Java Beans 3.0 <https://jcp.org/en/jsr/detail?id=220>, 09/02/2015
- [2] Javaworld: Understanding JPA, Part 1 <http://www.javaworld.com/article/2077817/java-se/understanding-jpa-part-1-the-object-oriented-paradigm-of-data-persistence.html>, 09/02/2015
- [3] Hibernate ORM project homepage <http://hibernate.org/orm/>, 07/27/2015
- [4] Hibernate Search project homepage <http://hibernate.org/search/>, 07/26/2015
- [5] Hibernate Search GitHub repository <https://github.com/hibernate/hibernate-search>, 07/26/2015
- [6] Hibernate Search FAQ <http://hibernate.org/search/faq/>, 08/27/2015
- [7] Lucene Website <https://lucene.apache.org/core/>, 07/16/2015
- [8] Elasticsearch Java API [<https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/index.html>], 07/27/2015
- [9] Solr Java API <https://wiki.apache.org/solr/Solrj>, 07/27/2015
- [10] xkcd #927 on competing standards <https://xkcd.com/927/>, 07/26/2015
- [11] Top-down programming, Robert Strandh <http://dept-info.labri.fr/~strandh/Teaching/PFS/Common/Strandh-Tutorial/top-down-programming.html>, 09/04/2015
- [12] Bottom-up programming, Robert Strandh <http://dept-info.labri.fr/~strandh/Teaching/PFS/Common/Strandh-Tutorial/bottom-up-programming.html>, 09/04/2015
- [13] objectmentor.com: Article on Single Responsibility Principle <http://www.objectmentor.com/resources/articles/srp.pdf>, 09/04/2015
- [14] objectmentor.com: Article on Open-Closed-Principle <http://www.objectmentor.com/resources/articles/ocp.pdf>, 09/04/2015
- [15] Object-Oriented Software Construction, Prentice Hall, 1988, Bertrand Meyer
- [16] Maven project homepage <https://maven.apache.org/>, 09/04/2015
- [17] Oracle JDBC overview <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>, 07/27/2015

- [18] Documentation on how to use OleDb with .NET [https://msdn.microsoft.com/en-us/library/5ybdbtte\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/5ybdbtte(v=vs.71).aspx), 07/27/2015
- [19] Wikibooks on Java Persistence https://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F, 07/27/2015
- [20] Stackoverflow JPA tag <http://stackoverflow.com/tags/jpa/info>, 09/04/2015
- [21] Hibernate OGM project homepage <http://hibernate.org/ogm/>, 07/27/2015
- [22] EclipseLink project homepage <http://www.eclipse.org/eclipselink/>, 07/27/2015
- [23] JSR 338: JPA 2.1 specification <https://jcp.org/en/jsr/detail?id=338>, 08/27/2015
- [24] OpenJPA project homepage <http://openjpa.apache.org/>, 07/27/2015
- [25] Java EE specification on oracle.com <http://www.oracle.com/technetwork/java/javaee/tech/index.html>, 09/04/2015
- [26] w3schools on SQL LIKE http://www.w3schools.com/sql/sql_like.asp, 07/27/2015
- [27] Lucene Tutorial <http://www.lucenetutorial.com/basic-concepts.html>, 07/20/2015
- [28] Elasticsearch Homepage <https://www.elastic.co/products/elasticsearch>, 08/19/2015
- [29] Solr Homepage <http://lucene.apache.org/solr/>, 08/19/2015
- [30] Elasticsearch Download website <https://www.elastic.co/downloads/elasticsearch>, 08/27/2015
- [31] Solr security <https://wiki.apache.org/solr/SolrSecurity>, 08/19/2015
- [32] elastic Shield (security for Elasticsearch) <https://www.elastic.co/products/shield>, 08/19/2015
- [33] Solr Administration (Core Specific Tools) <https://cwiki.apache.org/confluence/display/solr/Core-Specific+Tools>, 08/19/2015
- [34] ElasticHQ <http://www.elastichq.org/>, 08/19/2015
- [35] Elasticsearch: Life inside a cluster <https://www.elastic.co/guide/en/elasticsearch/guide/current/distributed-cluster.html>, 08/19/2015

- [36] Solr: Introduction to Scaling and Distribution <https://cwiki.apache.org/confluence/display/solr/Introduction+to+Scaling+and+Distribution>, 08/19/2015
- [37] hibernate-search-engine on mvnrepository.org <http://mvnrepository.com/artifact/org.hibernate/hibernate-search-engine/5.3.0.Final>, 08/27/2015
- [38] Hibernate Search roadmap <http://hibernate.org/search/roadmap/>, 08/14/2015
- [39] Hibernate Search documentation <http://hibernate.org/search/documentation/>, 07/31/2015
- [40] Hibernate Search documentation (MassIndexer, v5.4) https://docs.jboss.org/hibernate/search/5.4/reference/en-US/html_single/#search-batchindex-massindexer, 08/05/2015
- [41] MySQL - Developer's Library, Fourth Edition, 2009, Paul DuBois
- [42] CREATE TRIGGER in PostgreSQL <http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html>, 08/30/2015
- [43] Triggers in HSQLDB <http://hsqldb.org/doc/guide/triggers-chapt.html>, 08/30/2015
- [44] Triggers in Firebird <http://www.firebirdsql.org/refdocs/langrefupd21-ddl-trigger.html>, 08/30/2015
- [45] Truncate statement PostgreSQL docs <http://www.postgresql.org/docs/9.1/static/sql-truncate.html>, 08/27/2015
- [46] Firebird conformance <http://www.firebirdsql.org/en/sql-conformance/>, 08/27/2015
- [47] Hibernate Search GenericJPA GitHub repository <https://github.com/Hotware/Hibernate-Search-GenericJPA>, 08/13/2015
- [48] Hibernate Search JavaDoc <https://docs.jboss.org/hibernate/search/5.5/api/>, 09/08/2015

Erklärung

Erklärung zur Bachelorarbeit

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 :

