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 5, 2015
1. Supervisor:	Prof. Dr. Stefan Jablonski
2. Supervisor:	Prof. Dr. Bernhard Westfechtel

Zusammenfassung

Volltextsuchengines sind ein wertvolles Werkzeug um Suchergebnisse in Anwendungen zu verbessern, wenn relationale Datenbanken nicht ausreichen. Diese Engines sind nicht 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 und nicht mit anderen JPA konformen Providern, obwohl es möglich wäre dies zu erreichen. In dieser Thesis wird daher gezeigt, wie eine solche generische Version bewerkstelligt werden kann. Zuerst wird eine Einleitung in das Thema gegeben. Darauffolgend wird die verwendete Methodik erklärt. Danach wird eine kurzer Einblick über die verwendeten Technologien gegeben. Dann werden die größten Probleme beim Konstruieren der generischen Version aufgezeigt. Als nächstes wird eine Standalone Version von Hibernate Search beschrieben. Im Anschluss wird diese Standalone Version mit JPA integriert. Darauffolgend wird das Verhalten des automatischen Index updating Features der generischen Version von Hibernate Search beschrieben. Danach wird anhand eines Beispiels erklärt, wie die generische Version benutzt werden kann. Als Letztes wird ein Ausblick auf weitere Entwicklungsschritte nach der Thesis gegeben.

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 with the well established 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 and not with other JPA-conform providers even though it is possible to accomplish such a behaviour. In this thesis we will show how such a generic version can be accomplished. First we will give an introduction to the topic. Following, we will describe the methods we use. Subsequently, we will give a short overview over the technologies used. Then, we will describe the biggest challenges while building it. Next, we will work out a standalone version of Hibernate Search. After that we will integrate the standalone version with JPA. Following that, we will describe how the automatic index updating feature of our generic Hibernate Search works. Then, we will give a complete usage example of our generic version. Finally we will give an outlook on the development steps needed after this thesis.

Contents

1 Preface

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

The Java Persistence API (JPA) went even further by providing a standard 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 a 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. While all of them are JPA compliant (apart from minor bugs), some ship with additional modules to enhance their capabilities. A perfect example for this is the Hibernate Search API aimed at Hibernate ORM users.[3] [4]

[1] JSR 220: Enterprise Java Beans 3.0, see [1]
[2] Javaworld: Understanding JPA, Part 1, see [2]
[3] Hibernate ORM project homepage, see [3]
[4] Hibernate 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.

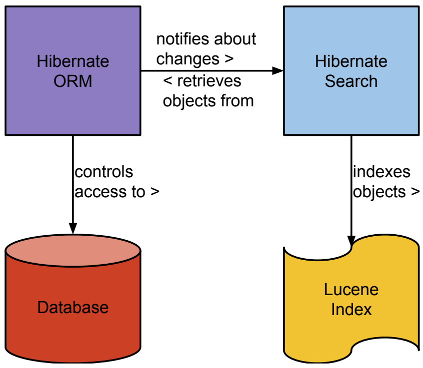


Figure 1: Hibernate Search with Hibernate ORM

Hibernate Search is based on the powerful Lucene search toolbox [5] [6] and is a separate project in the Hibernate family and aims to provide a JPA "feeling" in its API as it also incorporates a lot of JPA interfaces in its codebase. However, this does not mean that it is compatible with other JPA providers than Hibernate ORM (apart from Hibernate OGM, the NoSQL JPA mapper of the family) as the following figure 2 shows.

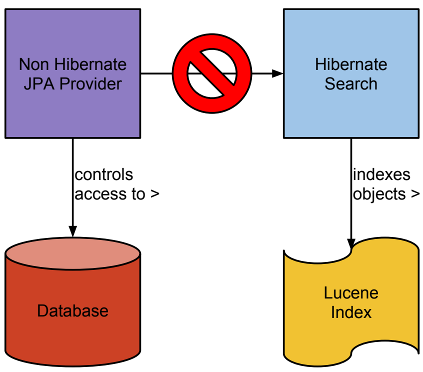


Figure 2: Hibernate Search's incompatibility with other JPA implementations

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.

[5] sourcecode on Hibernate Search GitHub, see [5]
[6] Hibernate 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 Lucene[7], ElasticSearch[8] or Solr[9]. While this is always a viable option, for some applications Hibernate Search would be a much better suit because of its design with a entity structure in mind and the automatic index updating feature, if it just were compatible with generic JPA.

When investigating Hibernate Search's project structure [10], we can see that the only module apart from some server-integration modules that depends on any ORM logic is "hibernate-search-orm". The modules that contain the indexing engine, the replication logic, alternative backends, etc. are completely independent from any ORM logic. This means, that most of the codebase could be reused for a generic version of Hibernate Search.

Creating 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 an all-purpose API for fulltext search in the JPA world instead of having a competing API that would just do the same thing in a different style.

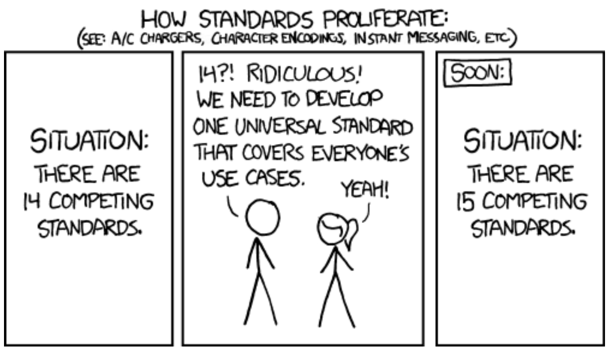


Figure 3: xkcd.com on competing standards [11]

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.

[7]official Lucene website, see [7]
[8]ElasticSearch Java API, see [8]
[9]Solr Java API, see [9]
[10]Hibernate Search GitHub repository, see [5]
[11]xkcd comic #927, see [10]

Short overview of contents:

In chapter 2 we explain what methods we are using to build Hibernate Search GenericJPA. In chapter 3 we give an overview over 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 a 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 we have achieved in this thesis and describe further steps.

2 Methods

For the development of the generic version of Hibernate Search we are using 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:

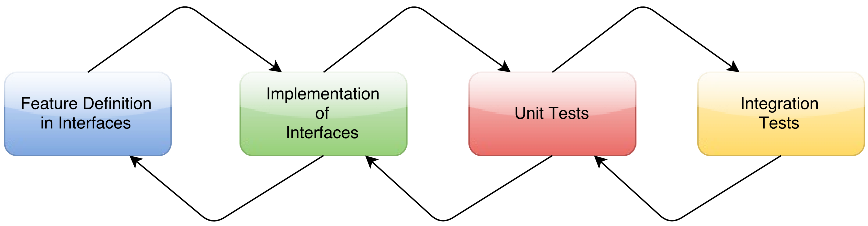


Figure 4: Development Process

- **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 explicitly 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 these according to the contracts set. As stated above, these classes are generally written against interfaces instead of direct implementations.

[12] Top-down programming, Robert Strandh, see [16]
[13] Bottom-up programming, Robert Strandh, see [17]
[14] objectmentor.com: Article on Single Responsibility Principle, see [12]
[15] objectmentor.com: Article on Open-Closed-Principle, see [13]
[16] Object-Oriented Software Construction, Prentice Hall, 1988, Bertrand Meyer, see [14]

- **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 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 (including the integration tests) are executed during each build to ensure no regression bugs occur. This is automatically managed by the Maven [17] build tool.

[17] Maven project homepage, see [15]

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, C#, etc. are object oriented[18]. While SQL solutions for querying relational databases exist for these languages (JDBC for Java[19], OleDb for C#[20]), 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.

```
SELECT author.id, author.name, book.id, book.name
FROM author_book, author, book
WHERE author_book.bookid = book.id
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 complex 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.

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

Listing 2: ORM query example

[18] Wikipedia on Object Oriented Programming (OOP), see [18]
[19] Oracle JDBC overview, see [19]
[20] OleDb usage page, see [20]

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"[21] [22]. While mostly known for standardizing relational database mappers (ORM), it also supports other concepts like NoSQL[23] [24] or XML storage[25]. However, when talking about JPA in this thesis, we will be focusing on the relational aspects of it. Currently, the newest version of this standard is 2.1 [26].

Some popular relational implementations are:

- Hibernate ORM (Red Hat)[27]
- EclipseLink (Eclipse foundation)[28]
- OpenJPA (Apache foundation)[29]

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

This is particularily important if you are working in a Java EE environment. Java EE itself is a specification for platforms, mostly Web-servers (JPA is part of the Java EE spec).[30] Many Java EE Web-servers ship with a bundled JPA implementation that they are optimized for (Wildfly with Hibernate ORM, GlassFish with EclipseLink, ...). This means that if the server is switched, it could also be a reasonable idea to swap out the JPA implementor. If everything in the application is written in a JPA compliant way, the user will then generally not run into many problems related to this switch.

[21] Wikibooks on Java Persistence, see [21]
[22] Stackoverflow JPA tag, see [22]
[23] Hibernate OGM project homepage, see [23]
[24] EclipseLink project homepage, see [24]
[25] EclipseLink project homepage, see [24]
[26] JSR 338: JPA 2.1 specification, see [25]
[27] Hibernate ORM project homepage, see [3]
[28] EclipseLink project homepage, see [24]
[29] OpenJPA project homepage, see [26]
[30] Java EE specification on oracle.com [27]

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 [31]:

```
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 since we would use vendor-specific features not every RDBMS supports.

Fulltext search engines can be used to complement databases in this regard. They are generally not intended to be replacing the database, but add additional functionality by indexing the data that is to be searched in a more sophisticated way. We will now take a look at some of the most popular available options for Java developers (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.

[31] w3schools on SQL LIKE, see [28]

3.3.1 Lucene

> Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.[32]

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 is 5.3.0 [33].

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 over 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 in[34], so it can be compared to a $Map < String, List < Text >>$ in Java. Before anything can be searched using Lucene, it has to be added to the the index (indexed) first.

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

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

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

[32] official Lucene website, see [7]
[33] official Lucene website, see [7]
[34] Lucene basic concepts, see [29]

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 has one additional problem: 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 Lucene based search servers available: ElasticSearch (by elastic)[35] and Solr (sister project of Lucene)[36]. Their current stable versions are 1.7.1 [37] and 5.3.0 [38] 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 [39] [40]. 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, ... [41] [42]). They are generally used because of their good clustering capabilities (distribution & replication) and are optimized for high throughput and scalability [43] [44]. 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).

[35] ElasticSearch Homepage, see [30]
[36] Solr Homepage, see [31]
[37] ElasticSearch Download website, see [32]
[38] Solr Homepage, see [31]
[39] Solr security, see [33]
[40] elastic Shield (security for ElasticSearch), see [34]
[41] Solr Administration (Core Specific Tools), see [35]
[42] ElasticHQ, see [36]
[43] ElasticSearch: Life inside a cluster, see [37]
[44] Solr: Introduction to Scaling and Distribution, see [38]

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. [45]

Hibernate Search's current stable version is 5.3.0.Final which is based on Lucene 4.10.4 [46].

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 and integrates with JPA interfaces seamlessly.

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 besides the already existing Lucene based backend and the optional Infinispan integration.

[45] Hibernate Search GitHub README, see [5]
[46] hibernate-search-engine on mvnrepository.org, see [39]

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.org [47]).

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 be used to inspire remodelling of the original Hibernate Search to incorporate generic JPA, which would probably 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".

[47] see `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. We will discuss the biggest ones after we have introduced a small example project first. This project will be used to showcase some problems and usages later on in this thesis as well.

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:

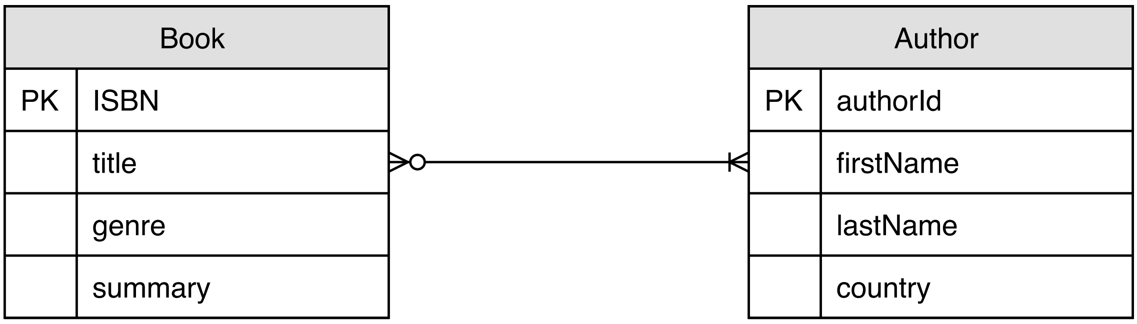


Figure 6: the bookstore entity relationship model

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 the following listings show.

```java
@Entity
@Table(name = "Book")
public class Book {

        @Id
        @Column(name = "isbn")
        private String isbn;

        @Column(name = "title")
        private String title;

        @Column(name = "genre")
        private String genre;

        @Lob
        @Column(name = "summary")
        private String summary;

        @ManyToMany(mappedBy = "books", cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
        private Set<Author> authors;

        //getters & setters ...
}
```

Listing 4: Book.java

```java
@Entity
@Table(name = "Author")
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "authorId")
        private Long authorId;

        @Column(name = "firstName")
        private String firstName;

        @Column(name = "lastName")
        private String lastName;

        @Column(name = "country")
        private String country;

        @ManyToMany(cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
        @JoinTable(name = "Author_Book",
                joinColumns =
                        @JoinColumn(name = "authorFk",
                                referencedColumnName = "authorId"),
                inverseJoinColumns =
                        @JoinColumn(name = "bookFk",
                                referencedColumnName = "isbn"))
        private Set<Book> books;

        //getters & setters ...
}
```

Listing 5: Author.java

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

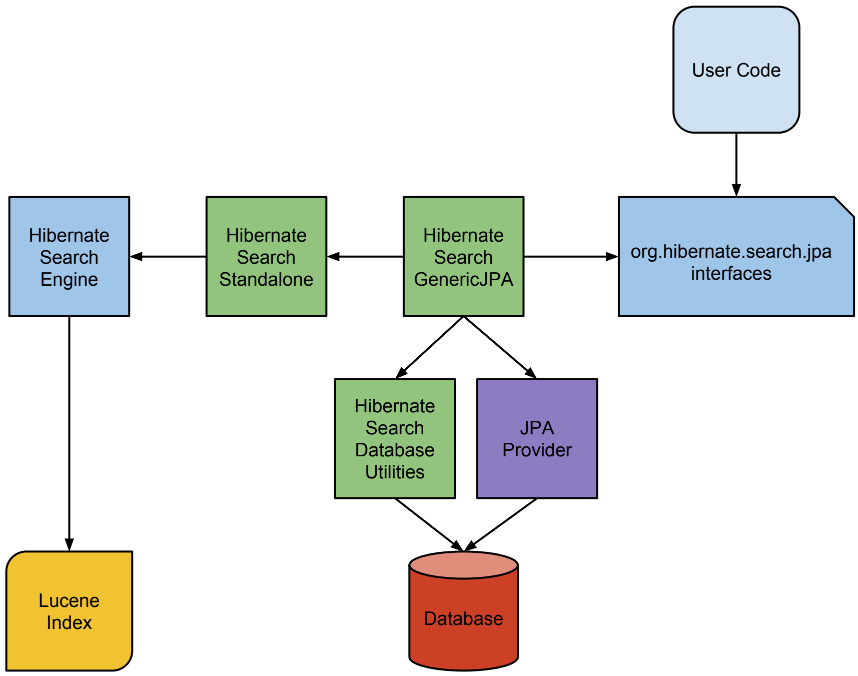


Figure 7: Complete Architecture of Hibernate Search GenericJPA

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 as 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 as 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:

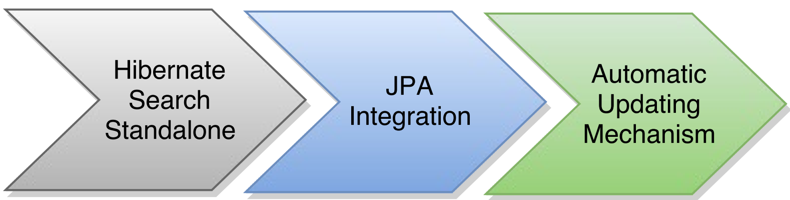
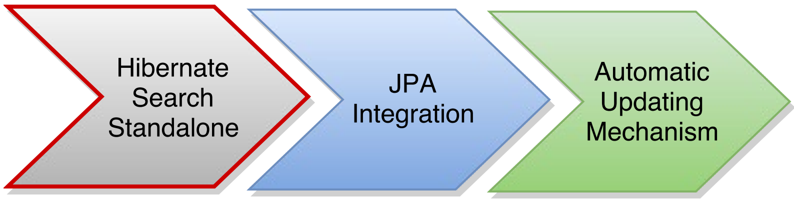


Figure 8: Timeline of the project

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 (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 (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.

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 which 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:

```
@Entity
@Table(name = "Book")
@Indexed
public class Book {

        @Id
        @Column(name = "isbn")
        @DocumentId
        private String isbn;

        @Column(name = "title")
        @Field(store = Store.YES, index = Index.YES)
        private String title;

        @Column(name = "genre")
        @Field(store = Store.YES, index = Index.YES)
        private String genre;

        @Lob
        @Column(name = "summary")
        @Field(store = Store.NO, index = Index.YES)
        private String summary;

        @ManyToMany(mappedBy = "books", cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
        @IndexedEmbedded(includeEmbeddedObjectId = true)
        private Set<Author> authors;

        //getters & setters ...
}
```

Listing 6: Book.java with Hibernate Search annotations

```java
@Entity
@Table(name = "Author")
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "authorId")
        @DocumentId
        private Long authorId;

        @Column(name = "firstName")
        @Field(store = Store.YES, index = Index.YES)
        private String firstName;

        @Column(name = "lastName")
        @Field(store = Store.YES, index = Index.YES)
        private String lastName;

        @Column(name = "country")
        @Field(store = Store.YES, index = Index.YES)
        private String country;

        @ManyToMany(cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
        @JoinTable(name = "Author_Book",
                joinColumns =
                        @JoinColumn(name = "authorFk",
                                referencedColumnName = "authorId"),
                inverseJoinColumns =
                        @JoinColumn(name = "bookFk",
                                referencedColumnName = "isbn"))
        @ContainedIn
        private Set<Book> books;

        //getters & setters ...
}
```

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 by Hibernate Search is "4.10.4". 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 [48] (only the Hibernate ORM specific settings cannot be used). Our **StandaloneSearchConfiguration** (appendix listing 44) defaults to "ram" and "4.10.4".

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

```
List<Class<?>> indexClasses = Arrays.asList(Book.class, Author.class);

SearchConfiguration searchConfiguration =
        new StandaloneSearchConfiguration();
indexClasses.forEach( searchConfiguration::addClass );

//bootstrapping class for Hibernate Search
SearchIntegratorBuilder builder = new SearchIntegratorBuilder();

//we have to build an integrator here (the builder needs a
//"base integrator" first before we can add index classes)
```

[48] Hibernate Search documentation, see [40]

```
builder.configuration( searchConfiguration ).buildSearchIntegrator();

indexClasses.forEach( builder::addClass );

//starts the engine with all configuration properties set
SearchIntegrator searchIntegrator = builder.buildSearchIntegrator();

//use the integrator ...

//close it
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**:

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

Listing 9: Indexing an object with the engine

Updating objects with **WorkType.UPDATE**:

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

Listing 10: Updating an object with the engine

Deleting objects with **WorkType.PURGE**:

```
String isbn = ...;
Transaction tx = new Transaction();
Worker worker = searchIntegrator.getWorker();
worker.performWork( new Work( Book.class, isbn, WorkType.PURGE ), tx );
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 used in Hibernate Search ORM is available (the Builder returns a Lucene query. Any basic Lucene query could be used as well, but require manual analysis of the input. Queries produced by the builder are automatically analysed with the correct Analyzer).

```java
SearchIntegrator searchIntegrator = ...;

HSQuery query = searchIntegrator.createHSQuery();

//find information about all the entities matching a given title
List<EntityInfo> entityInfos =
        query.luceneQuery(
                        //query DSL:
                        searchIntegrator.buildQueryBuilder()
                                .forEntity( Book.class )
                                .get()
                                .keyword()
                                .onField( "title" )
                                .matching( "searchString" )
                                .createQuery()
                ).targetedEntities(
                        Collections.singletonList(
                                Book.class
                        )
                ).projection(
                        ProjectionConstants.ID
                ).queryEntityInfos();
```

Listing 12: Querying the index with the engine

The queries don'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 the example above 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.

```
//a JPA EntityManager
EntityManager em = ...;

//extract info from the entityInfos
for(EntityInfo entityInfo : entityInfos) {
        String isbn = (String) entityInfo.getProjection()[0];
        //retrieve an object from the database
        Book book = em.find(Book.class, isbn);
        //handle this information ...
}
```

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 class 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.

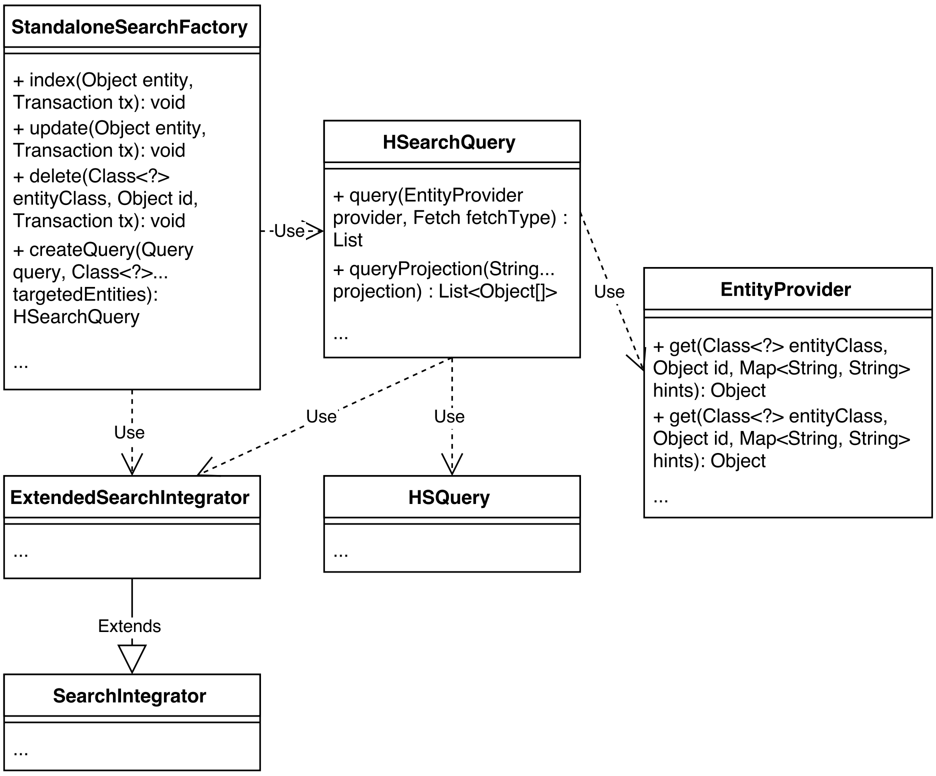


Figure 9: Rough architecture of the standalone version (important parts)

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.

```
List<Class<?>> indexClasses = Arrays.asList(Book.class, Author.class);

//we still have to build the SearchConfiguration object
SearchConfiguration searchConfiguration =
                new StandaloneSearchConfiguration();
indexClasses.forEach( searchConfiguration::addClass );

//the builder pattern from before is abstracted in the following lines
StandaloneSearchFactory searchFactory =
                StandaloneSearchFactoryFactory.
                                createSearchFactory(
                                        searchConfiguration,
                                        indexClasses
                                );

//use the searchfactory ...

//close it
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 SearchIntegrator's Worker pattern anymore.

```
Book book = ...;
Transaction tx = new Transaction();
searchFactory.index(book, tx);
tx.commit();
```

Listing 15: Indexing an object with the standalone version

```
Book book = ...;
Transaction tx = new Transaction();
searchFactory.update(book, tx);
tx.commit();
```

Listing 16: Updating an object with the standalone version

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

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

5.3.3 Queries

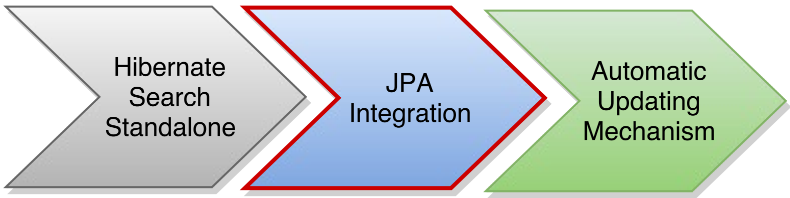
The biggest change in the standalone version is probably how the index is queried. We don't have to work with EntityInfos anymore as we introduced the **EntityProvider** interface. This interface hosts one method that is to be used for batch fetching (Fetch.BATCH) and one for single fetching (Fetch.FIND_BY_ID).

A good default implementation delegating the database access to a JPA EntityManager is our **BasicEntityProvider** (45). Besides taking an EntityManager in its constructor, the class 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 example for the sake of simplicity, the code for this can be found in the listings (46). After its creation this map can then be stored in a central place and reused.

```
StandaloneSearchFactory searchFactory = ...;

EntityManager em = ...;
Map<Class<?>, String> idProperties = ...;

EntityProvider entityProvider = new BasicEntityProvider(em, idProperties);

List<Book> books = searchFactory
                    .createQuery(searchFactory.buildQueryBuilder()
                            .forEntity(Book.class)
                            .get()
                            .keyword()
                            .onField("title")
                            .matching("searchString")
                            .createQuery(), Book.class
                    ).query(
                            entityProvider,
                            Fetch.BATCH
                    );
```

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) - which was one of the reasons why we chose to use Hibernate Search's engine in the first place - 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 point 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 adding 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.

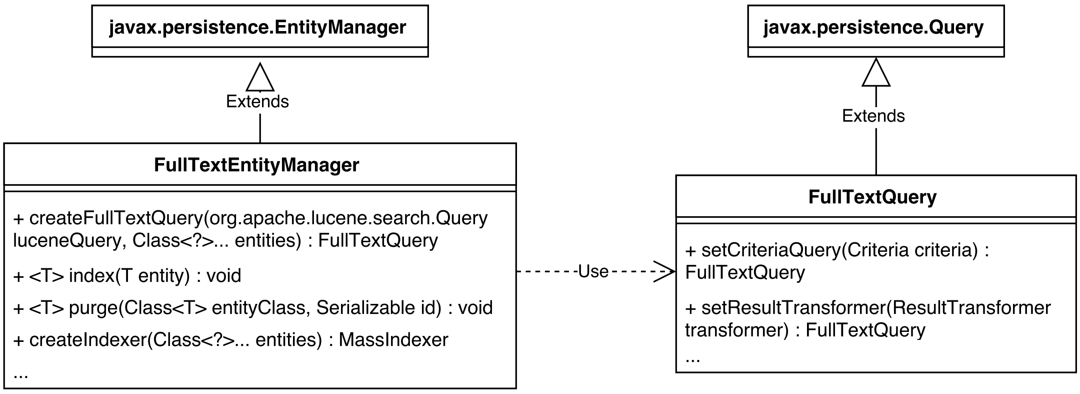


Figure 10: The main JPA interfaces of Hibernate Search ORM

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:

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

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:

```
EntityManager em = ...;
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 5.3 to work like apart from minor naming differences.

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

```
FullTextEntityManager fem = ...;
Book book = ...;
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(...).

```
FullTextEntityManager fem = ...;
String isbn = ...;
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.

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

FullTextQuery fullTextQuery = fem.createFullTextQuery(
        fem.getSearchFactory().buildQueryBuilder()
                .forEntity( Book.class )
                .get()
                .keyword()
                .onField( "title" )
                .matching( "searchString" )
                .createQuery(),
        Book.class);

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:

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

fem.createIndexer( Book.class )
        .batchSizeToLoadObjects( 25 )
        .threadsToLoadObjects( 12 )
        .idFetchSize( 150 )
        .transactionTimeout( 1800 )
        .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."[49]

[49]Hibernate Search documentation (MassIndexer, v5.4), see [41]

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.

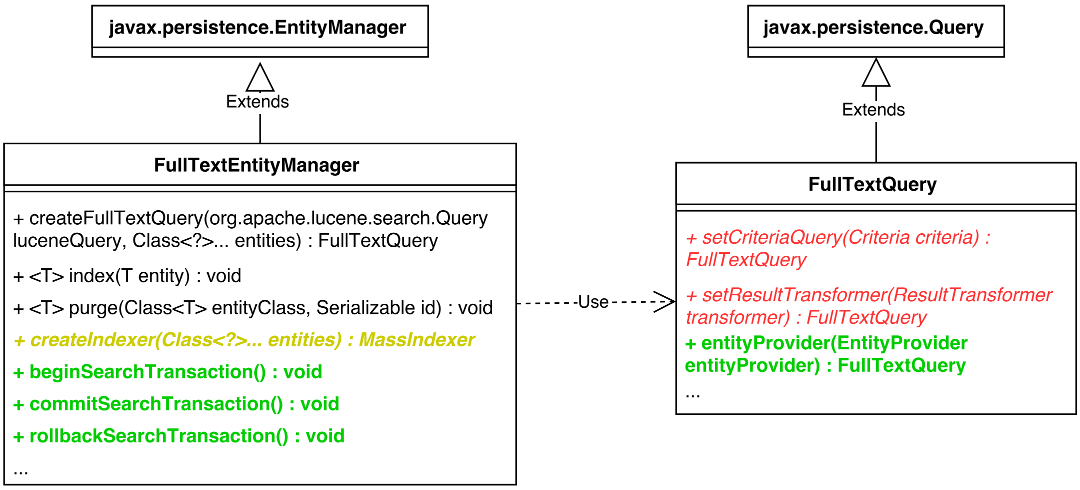


Figure 11: Required fixes for a generic version

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 discuss 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 **JPASearch-FactoryController**:

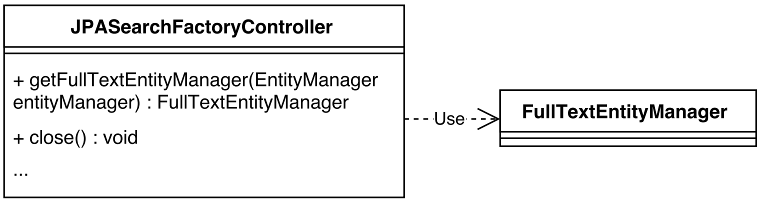


Figure 12: JPASearchFactoryController

Having this separate interface means that its lifecycle has to be controlled on its own.

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 works in Hibernate Search ORM only 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:

```
//In Hibernate Search ORM, the fulltext engine would be started
//together with the EntityManagerFactory.
//In GenericJPA we can't do that.
EntityManagerFactory emf = ...;

EntityManager em = ...;

Properties properties = new Properties();

properties.setProperty(
        "hibernate.search.searchfactory.type",
        "manual-updates"
);

//In GenericJPA this starts the fulltext engine to
//which a reference is returned by this method call
JPASearchFactoryController searchFactoryController =
        Setup.createSearchFactoryController( emf, properties );

//FullTextEntityManagers are not obtained with the Search class
FullTextEntityManager fem =
        searchFactoryController.getFullTextEntityManager(em);

//use it...

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 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 manipulation is 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:

```
EntityManager em = ...;
JPASearchFactoryController searchFactoryController = ...;

FullTextEntityManager fem =
        searchFactoryController.getFullTextEntityManager(em);

fem.beginSearchTransaction();
try {
        //index or purge here
        fem.commitSearchTransaction();
} catch(Exception e) {
        fem.rollbackSearchTransaction();
        throw e;
}
```

Listing 26: Index control with Hibernate Search GenericJPA

By introducing our own search transaction management methods we don't restrict the usage of GenericJPA in application servers by a lot compared to the original Hibernate Search ORM because manual index changes are not needed frequently. In general these transactions can not be compared with real RDBMS transactions anyways as it is allowed to write changes to the index without commiting with flushToIndexes(). These changes 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, etc.. But the engine needs to know which class to get the index description metamodel from.

This is the reason in Hibernate Search GenericJPA we implement logic to feed the right entity class into the engine via user input. 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 the a best effort as this is the behaviour Hibernate Search ORM has. This algorithm is described in Java code in the next listing 27.

```
//get the first class in the hierarchy
Class<T> clazz = (Class<T>) entity.getClass();

//check if the original class has @InIndex present
//if yes, we don't have to go higher up in the class hierarchy
if ( !clazz.isAnnotationPresent( InIndex.class ) ) {

	//go up in the class hierarchy until either a @InIndex is found
	//or there is no superclass anymore.
	while ( (clazz = (Class<T>) clazz.getSuperclass()) != null ) {
		if ( clazz.isAnnotationPresent( InIndex.class ) ) {
			break;
		}
	}

}

//if we have found a class annotated with @InIndex
//we return it here
if ( clazz != null ) {
	return clazz;
}

//no @InIndex found, try the entities direct class
//as a best effort
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:

```
@Entity
@InIndex
@Table(name = "Book")
@Indexed
public class Book {

        //rest is unchanged

}
```

Listing 28: Book.java with @InIndex

```
@Entity
@InIndex
@Table(name = "Author")
public class Author {

        //rest is unchanged

}
```

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. In the future a hybrid approach checking for both annotations is possible, but using only @InIndex is sufficient.

6.3.3 Queries

While we didn't mention this in 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 when the data is actually 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 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.

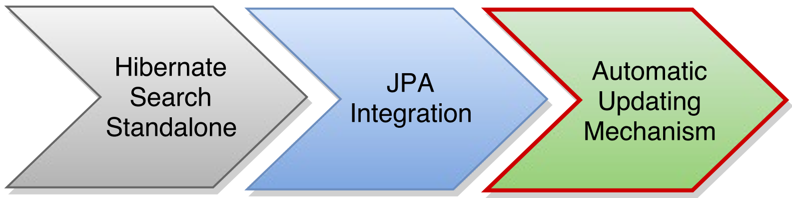
The basic ideas are the same though: Each entity type has its ids scrolled from the database by one thread (there can be multiple threads doing this, but for other entities) and then a configurable amount of indexing threads handles these ids batch by batch in a Hibernate Search index-writing backend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

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

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

fem.createIndexer( Book.class )
        .batchSizeToLoadObjects( 25 )
        .threadsToLoadObjects( 12 )
        .batchSizeToLoadIds( 150 )
        .idProducerTransactionTimeout( 1800 )
        .startAndWait();
```

Listing 30: MassIndexer usage with Hibernate Search ORM

7 Automatic index updating



As already stated in 4.4, the automatic index updating feature is a 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 comparison 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 sync and async, while the async version will serve as a backup whenever the synchronized 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 looks 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 class doesn't have to implement an interface, but has to 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.

Such a class generally looks like this:

```
public class EntityListener {

        @PostPersist
        public void persist(Object entity) {
                //handle the event
        }

        @PostUpdate
        public void update(Object entity) {
                //handle the event
        }

        @PostDelete
        public void delete(Object entity) {
                //handle the event
        }

}
```

Listing 31: Example JPA entity listener

This EntityListener is then applied with an annotation on the entity:

```
@EntityListeners( { EntityListener.class } )
public class Book {

        //...

}
```

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.

```
public class EntityListener {

        public EntityListener() {
                // inject it somewhere
                // so we can access it in a static way
                EntityListenerRegistry.inject(this);
        }

        //...

}
```

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 can see that the events are triggered on flush instead of commit. This is an issue if the changed data is not actually commited.

```
EntityManager em = ...;

em.getTransaction().begin();

Book book = em.find( Book.class "someIsbn" );
book.setTitle( "someNewTitle" );

// flushes, so we retrieve the Book with the changes from above
// => event is triggered
List<Book> allBooks =
        em.createQuery( "SELECT b FROM Book b" ).getResultList();

// we have no way to get this event to revert the wrong index change
em.getTransaction().rollback();
```

Listing 34: Event triggering on flush

While it **might** be possible to somehow fix the flush issue, the bad support from JPA providers like Hibernate ORM renders this approach unusable until the JPA providers work the same way 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)

They generally have callbacks similar to these 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 async 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 [...]. [50]

While the quote above is meant to be for MySQL databases, many other RDBMS support at least triggers on the three crucial events for event-listening: INSERT (CREATE), UPDATE, DELETE, just like MySQL [51] [52] [53].

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 - by definition transactional - triggers, we don't have to 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.

[50] MySQL - Developer's Library, see [42]
[51] CREATE TRIGGER in PostgreSQL, see [43]
[52] Triggers in HSQLDB, see [44]
[53] Triggers in Firebird, see [45]

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.

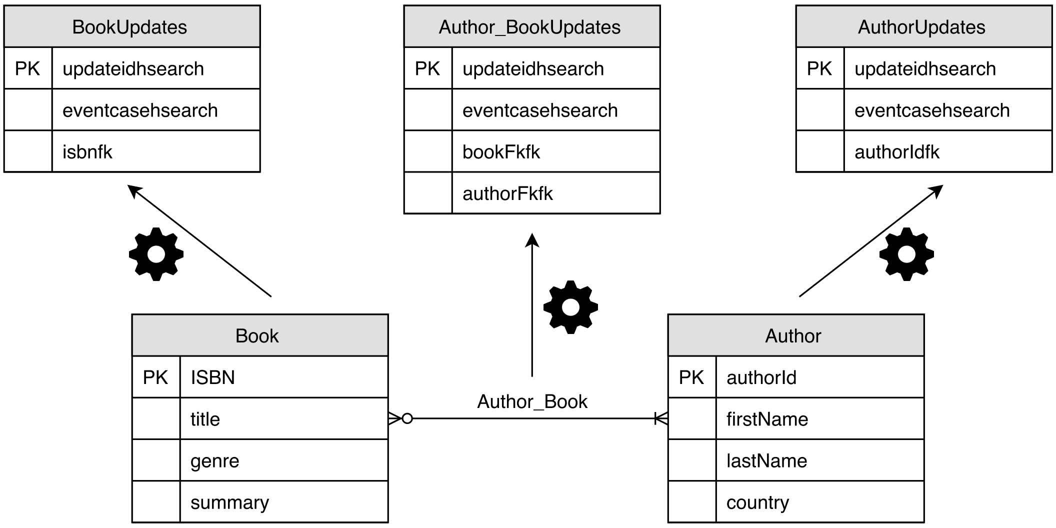


Figure 13: Triggers for the example project

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 occured. 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 could be achieved 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, ...). While working out a correct solution for all the different cases would be possible, it would be quite time-consuming and wouldn't have fit in the time schedule of this thesis. However, this does not mean that we won't try to build it in the future.

The following listings show the @UpdateInfo annotation in use:

```
@Entity
@InIndex
@Table(name = "Book")
@Indexed
@UpdateInfo(
        tableName = "Book",
        idInfos = @IdInfo(
                columns = @IdColumn(
                        column = "isbn",
                        columnType = ColumnType.STRING
                )
        )
)
public class Book {

        // ... unchanged.

        // mapping table events handled on Author side

        // getters & setters ...
}
```

Listing 35: Book.java with Hibernate Search annotations

```java
@Entity
@InIndex
@Table(name = "Author")
@UpdateInfo(
        tableName = "Author",
        idInfos = @IdInfo(
                columns = @IdColumn(
                        column = "authorId",
                        columnType = ColumnType.LONG
                )
        )
)
public class Author {

        // ... unchanged.

        @UpdateInfo(tableName = "Author_Book",
                idInfos = {
                @IdInfo(entity = Author.class,
                        columns = @IdColumn(
                                column = "authorFk",
                                columnType = ColumnType.LONG
                        )
                ),
                @IdInfo(entity = Book.class,
                        columns = @IdColumn(
                                column = "bookFk",
                                columnType = ColumnType.STRING
                        )
                )
        })
        private Set<Book> books;

        //getters & setters ...
}
```

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

However, if the developer needs different names in the update tables (e.g. if there already exists a table with the same name), it is possible to manually set these properties. They can be found on the same level 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 have to 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 HSQDLB. See table 2 for information about how to set the correct one for each database.

Whether and how the triggers and tables are generated at all can also be set, but with a configuration property on the SearchFactoryController as described in table 2. If 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 now 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:

```
SELECT t1.updateidhsearch, t1.authorFkfk, t1.bookFkfk
FROM author_bookupdates t1
INNER JOIN
(
        /* select the most recent update */
        SELECT max(t2.updateidhsearch) updateid,
                t2.authorFkfk, t2.bookFkfk
        FROM author_bookupdates t2
        GROUP BY t2.authorFkfk, t2.bookFkfk
) t3 on t1.updateidhsearch = t3.updateid
/* handle events that occured earlier first */
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 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 have to enforce **@IndexedEmbedded#includeEmbeddedObjectId = true**. 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.

Note that 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 [54], which some RDBMSs don't fully support [55]. 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).

```
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.

[54]Truncate statement PostgreSQL docs, see [46]
[55]Firebird conformance, see [47]

7.2 Comparison of approaches

We already discussed the differences of synchronous and asynchronous approaches in general earlier this chapter. 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.

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 as shown in 14. This is the reason the user has to add this information as we have seen in 7.1.2.2.

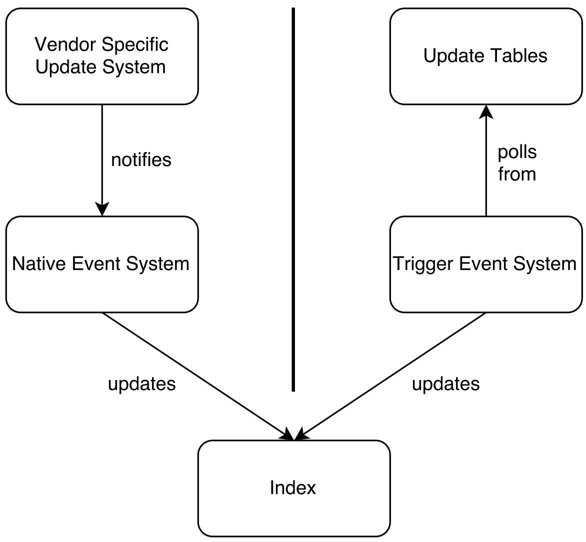


Figure 14: Hibernate Search GenericJPA update mechanisms

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. This includes native SQL queries from EntityManagers. This also means that the database can only be used by the JPA application and no other scripts, small programs etc. 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 Conclusion

We can see that both event systems can be useful in different cases. This is the reason we use both in Hibernate Search GenericJPA. The following table summarizes the pros and cons once again:

Approach	Pros	Cons
Native Event System	+ No additional work needed by the developer	- 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	+ Works with any JPA implementation (even rarely used ones) + Changes from outside of the JPA provider are recognized (e.g. native SQL access)	- Additional work by the developer needed (annotations)

Table 1: Pros and Cons of the two update systems

8 Usage of Hibernate Search GenericJPA

Having described how Hibernate Search GenericJPA works and is designed we will now take a look at how it can be used in our example project (4.1). While having already explained this part by part in each chapter, the following is everything put together and using the async updating mechanism as described in 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 known from the original Hibernate Search ORM (5.1). Using Hibernate Search GenericJPA then requires us also to add the @InIndex on every entity contained in the index (6.3.2) and because we are using the async updating mechanism here, we have to add information about how to create the update tables as well (7.1.2.2).

The resulting entities with the changes highlighted look like this:

```
@Entity
@Table(name = "Book")
@InIndex
@Indexed
@UpdateInfo(tableName = "Book",
        idInfos = @IdInfo(
                columns = @IdColumn(
                        column = "isbn",
                        columnType = ColumnType.STRING)))
public class Book {

        @Id
        @DocumentId
        @Column(name = "isbn")
        private String isbn;

        @Column(name = "title")
        @Field
        private String title;

        @Column(name = "genre")
        @Field
        private String genre;

        @Lob
        @Column(name = "summary")
        @Field
        private String summary;

        @ManyToMany(mappedBy = "books", cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
```

```
        @IndexedEmbedded(includeEmbeddedObjectId = true)
        private Set<Author> authors;

        // getters & setters ...

}
```

Listing 39: Book.java complete

```
@Entity
@Table(name = "Author")
@InIndex
@UpdateInfo(tableName = "Author",
        idInfos = @IdInfo(
                columns = @IdColumn(
                        column = "authorId",
                        columnType = ColumnType.LONG
        )
))
public class Author {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "authorId")
        @DocumentId
        private Long authorId;

        @Column(name = "firstName")
        @Field
        private String firstName;

        @Column(name = "lastName")
        @Field
        private String lastName;

        @Column(name = "country")
        @Field
        private String country;

        @ManyToMany(cascade = {
                CascadeType.MERGE,
                CascadeType.DETACH,
                CascadeType.PERSIST,
                CascadeType.REFRESH
        })
        @JoinTable(name = "Author_Book",
                joinColumns = @JoinColumn(name = "authorFk",
                        referencedColumnName = "authorId"),
                inverseJoinColumns = @JoinColumn(name = "bookFk",
```

```
                        referencedColumnName = "isbn"))
        @UpdateInfo(tableName = "Author_Book",
                idInfos = {
                        @IdInfo(entity = Author.class,
                                columns = @IdColumn(
                                column = "authorFk",
                                columnType = ColumnType.LONG)),
                        @IdInfo(entity = Book.class,
                                columns = @IdColumn(
                                column = "bookFk",
                                columnType = ColumnType.STRING))
        })
        @ContainedIn
        private Set<Book> books;

        // getters & setters ...

}
```

Listing 40: Author.java complete

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.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0">

        <persistence-unit name="EclipseLink_HSQLDB"
                transaction-type="RESOURCE_LOCAL">
                <provider>
                        org.eclipse.persistence.jpa.PersistenceProvider
                </provider>
                <class>*.*.Author</class>
                <class>*.*.Book</class>
                <properties>
                        <property name="javax.persistence.jdbc.driver"
                                value="org.hsqldb.jdbcDriver"/>
                        <property name="javax.persistence.jdbc.url"
                                value="jdbc:hsqldb:mem:test"/>
                        <property name="javax.persistence.jdbc.user"
                                value="user"/>
                        <property name="javax.persistence.jdbc.password"
                                value="password"/>
                        <property name="eclipselink.ddl-generation"
                                value="drop-and-create-tables"/>
                        <property name="eclipselink.logging.level"
                                value="INFO"/>
                        <property name="eclipselink.ddl-
                                generation.output-mode"
                                value="both"/>
                </properties>
        </persistence-unit>

</persistence>
```

Listing 41: persistence.xml complete

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.

```
Properties properties = new Properties();

// use the async backend
properties.setProperty(
        "hibernate.search.searchfactory.type",
        "sql"
);

// we are using HSQLDB, so use the right TriggerSource
properties.setProperty(
        "hibernate.search.trigger.source",
        "org.hibernate.search.genericjpa.db." +
                "events.triggers.HSQLDBTriggerSQLStringSource"
);

// start up the EntityManagerFactory (entry-point to JPA)
// and create one EntityManager
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory( "EclipseLink_HSQLDB" );
EntityManager em = emf.createEntityManager();

// start up Hibernate Search GenericJPA
JPASearchFactoryController searchController =
        Setup.createSearchFactoryController( emf, properties );

// persist entities in the database
em.getTransaction().begin();
Author author = ...;
Book book = ...;
book.setAuthor( author );
em.persist( em );
em.getTransaction().commit();

// we are using an async backend, so wait a bit
// for the updating mechanism to handle the
// persist (Exception not handled here)
Thread.sleep( 10_000 );

// create a FullTextEntityManager
FullTextEntityManager fem = searchController
        .getFullTextEntityManager( em );

```

```
// query for all Books having the title "searchString"
FullTextQuery fullTextQuery = fem.createFullTextQuery(
        fem.getSearchFactory().buildQueryBuilder()
                .forEntity( Book.class )
                .get()
                .keyword()
                .onField( "title" )
                .matching( "searchString" )
                .createQuery(),
        Book.class);

List<Book> books = (List<Book>) fullTextQuery.getResultList();

//handle the books
System.out.println( books );

// close everything
// (FullTextEntityManager is not closed because
// the EntityManager is closed)
em.close();
searchController.close();
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 similar 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. At least some of the info is known to be able to be retrieved from the JPA provider (via specific interfaces, Hibernate ORM and EclipseLink definitely have this option, others have not been checked). These automatic mechanisms could not be included in this thesis as they would have required a lot more time than was available.

During the process of designing and writing the code for Hibernate Search GenericJPA we tried to be as compatible with the orginal 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 even though the code as it can be found on GitHub [56] can already be used in real applications. In fact as described in chapter 2 every relevant part of Hibernate Search GenericJPA has been extensively tested in single feature-tests and integration-tests and 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 the merging process 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 and [57] 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 can be looked at as a de-facto standard for fulltext search in JPA. Having such a standard would be quite beneficial for the ever changing JPA world as smaller JPA providers could have a better chance at getting a bigger user base, which is good for research and innovation.

[56] Hibernate Search GenericJPA GitHub repository, see [48]
[57] Hibernate Search roadmap, see [49]

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 [58]):

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

[58]Hibernate Search GenericJPA GitHub repository, see [48]

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.

```java
public class Transaction implements TransactionContext {

	private boolean progress = true;
	private List<Synchronization> syncs = new ArrayList<>();

	@Override
	public boolean isTransactionInProgress() {
		return this.progress;
	}

	@Override
	public Object getTransactionIdentifier() {
		return this;
	}

	@Override
	public void registerSynchronization(
		Synchronization synchronization ) {
		this.syncs.add( synchronization );
	}

	/**
	 * @throws IllegalStateException if already commited/rolledback
	 */
	public void commit() {
		if ( !this.progress ) {
			throw new IllegalStateException(
			"can't commit - " +
			"No Search Transaction is in Progress!" );
		}
		this.progress = false;
		this.syncs.forEach( Synchronization::beforeCompletion );

		for ( Synchronization sync : this.syncs ) {
			sync.afterCompletion( Status.STATUS_COMMITTED );
		}
```

```java
        }

        /**
         * @throws IllegalStateException if already commited/rolledback
         */
        public void rollback() {
                if ( !this.progress ) {
                        throw new IllegalStateException(
                        "can't rollback - " +
                        "No Search Transaction is in Progress!" );
                }
                this.progress = false;
                this.syncs.forEach( Synchronization::beforeCompletion );

                for ( Synchronization sync : this.syncs ) {
                        sync.afterCompletion( Status.STATUS_ROLLEDBACK );
                }
        }

}
```

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.

```
/**
 * Manually defines the configuration.
 * Classes and properties are the only implemented options at the moment.
 *
 * @author Martin Braun (adaption), Emmanuel Bernard
 */
public class StandaloneSearchConfiguration
        extends SearchConfigurationBase
        implements SearchConfiguration {

        private final Logger LOGGER =
                Logger.getLogger(
                        StandaloneSearchConfiguration.class.getName()
                );

        private final Map<String, Class<?>> classes;
        private final Properties properties;
        private final HashMap<Class<? extends Service>, Object>
                providedServices;
        private final InstanceInitializer initializer;
        private SearchMapping programmaticMapping;
        private boolean transactionsExpected = true;
        private boolean indexMetadataComplete = true;
        private boolean idProvidedImplicit = false;
        private ClassLoaderService classLoaderService;
        private ReflectionManager reflectionManager;

        public StandaloneSearchConfiguration() {
                this( new Properties() );
        }

        public StandaloneSearchConfiguration(Properties properties) {
                this(
                        SubClassSupportInstanceInitializer.INSTANCE,
                        properties
                );
        }

        public StandaloneSearchConfiguration(InstanceInitializer init) {
                this( new Properties() );
        }

```

```
	public StandaloneSearchConfiguration(InstanceInitializer init,
			Properties properties) {
		this.initializer = init;
		this.classes = new HashMap<>();
		this.properties = properties;
		// default values if nothing was explicitly set
		this.properties.computeIfAbsent(
				"hibernate.search.default.directory_provider",
				(key) -> {
					LOGGER.info(
					  "defaulting to RAM directory-provider"
					);
				return "ram";
		});
		this.properties.computeIfAbsent(
				"hibernate.search.lucene_version",
				(key) -> {
					LOGGER.info(
							"defaulting to Lucene Version: "
							+ Version.LUCENE_4_10_4.toString()
					);
					return Version.LUCENE_4_10_4.toString();
		});
		this.reflectionManager = new JavaReflectionManager();
		this.providedServices = new HashMap<>();
		this.classLoaderService = new DefaultClassLoaderService();
	}

	public StandaloneSearchConfiguration addProperty(String key,
			String value) {
		properties.setProperty( key, value );
		return this;
	}

	public StandaloneSearchConfiguration addClass(Class<?> indexed) {
		classes.put( indexed.getName(), indexed );
		return this;
	}

	@Override
	public Iterator<Class<?>> getClassMappings() {
		return classes.values().iterator();
	}

	@Override
	public Class<?> getClassMapping(String name) {
```

```
                return classes.get( name );
        }

        @Override
        public String getProperty(String propertyName) {
                return properties.getProperty( propertyName );
        }

        @Override
        public Properties getProperties() {
                return properties;
        }

        @Override
        public ReflectionManager getReflectionManager() {
                return this.reflectionManager;
        }

        @Override
        public SearchMapping getProgrammaticMapping() {
                return programmaticMapping;
        }

        public StandaloneSearchConfiguration setProgrammaticMapping(
                        SearchMapping programmaticMapping
                ) {
                this.programmaticMapping = programmaticMapping;
                return this;
        }

        @Override
        public Map<Class<? extends Service>, Object>
                getProvidedServices() {
                return providedServices;
        }

        public void addProvidedService(
                        Class<? extends Service> serviceRole,
                        Object service
                ) {
                providedServices.put( serviceRole, service );
        }

        @Override
        public boolean isTransactionManagerExpected() {
                return this.transactionsExpected;
        }

```

```
	public void setTransactionsExpected(
				boolean transactionsExpected) {
		this.transactionsExpected = transactionsExpected;
	}

	@Override
	public InstanceInitializer getInstanceInitializer() {
		return initializer;
	}

	@Override
	public boolean isIndexMetadataComplete() {
		return indexMetadataComplete;
	}

	public void setIndexMetadataComplete(
		boolean indexMetadataComplete) {
		this.indexMetadataComplete = indexMetadataComplete;
	}

	@Override
	public boolean isIdProvidedImplicit() {
		return idProvidedImplicit;
	}

	public StandaloneSearchConfiguration
		setIdProvidedImplicit(boolean idProvidedImplicit) {
		this.idProvidedImplicit = idProvidedImplicit;
		return this;
	}

	@Override
	public ClassLoaderService getClassLoaderService() {
		return classLoaderService;
	}

	public void setClassLoaderService(
		ClassLoaderService ) {
		this.classLoaderService = classLoaderService;
	}

}
```

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.

```
public class BasicEntityProvider implements EntityProvider {

        private static final String QUERY_FORMAT =
                "SELECT obj FROM %s obj " +
                "WHERE obj.%s IN :ids";
        private final EntityManager em;
        private final Map<Class<?>, String> idProperties;

        public BasicEntityProvider(EntityManager em,
                Map<Class<?>, String> idProperties) {
                this.em = em;
                this.idProperties = idProperties;
        }

        @Override
        public void close() throws IOException {
                this.em.close();
        }

        @Override
        public Object get(Class<?> entityClass, Object id,
                Map<String, String> hints) {
                return this.em.find( entityClass, id );
        }

        @SuppressWarnings({"rawtypes", "unchecked"})
        @Override
        public List getBatch(Class<?> entityClass, List<Object> ids,
                Map<String, String> hints) {
                List<Object> ret = new ArrayList<>( ids.size() );
                if ( ids.size() > 0 ) {
                        String idProperty =
                                this.idProperties.get( entityClass );
                        String queryString =
                                String.format(
                                        QUERY_FORMAT,
                                        this.em.getMetamodel()
                                                .entity( entityClass )
                                                .getName(),
                                        idProperty
                );
                Query query = this.em.createQuery( queryString );
```

```
            query.setParameter( "ids", ids );
                ret.addAll( query.getResultList() );
            }
            return ret;
        }

        public void clearEm() {
            this.em.clear();
        }

        public EntityManager getEm() {
            return this.em;
        }

}
```

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.

```
SearchConfiguration config = ...;

MetadataProvider metadataProvider =
        MetadataUtil.getDummyMetadataProvider( config );
MetadataRehasher rehasher = new MetadataRehasher();

List<RehashedTypeMetadata> rehashedTypeMetadatas = new ArrayList<>();
for ( Class<?> indexRootType : this.getIndexRootTypes() ) {
        RehashedTypeMetadata rehashed =
                rehasher.rehash(
                        metadataProvider
                                .getTypeMetadataFor( indexRootType )
                );
        rehashedTypeMetadatas.add( rehashed );
}

Map<Class<?>, String> idProperties =
        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.

```java
/**
 * Utility class that allows you to access multiple JPA queries at once.
 * Data is retrieved from the database in batches
 * and ordered by a given comparator.
 * No need for messy Unions on the database level! <br>
 * <br>
 * This is particularly useful if you scroll all the data
 * from the database incrementally and if you can
 * compare in Code.
 *
 * @author Martin
 */
public class MultiQueryAccess {

        private final Map<String, Long> currentCountMap;
        private final Map<String, Query> queryMap;
        private final Comparator<ObjectIdentifierWrapper> comparator;
        private final int batchSize;

        private final Map<String, Long> currentPosition;
        private final Map<String, LinkedList<Object>> values;

        private Object scheduled;
        private String identifier;


        public MultiQueryAccess(
                Map<String, Long> countMap,
                Map<String, Query> queryMap,
                Comparator<ObjectIdentifierWrapper> comparator,
                int batchSize) {
                if ( countMap.size() != queryMap.size() ) {
                        throw new IllegalArgumentException(
                                "countMap.size() must be equal " +
                                        "to queryMap.size()" );
                }
                this.currentCountMap = countMap;
                this.queryMap = queryMap;
                this.comparator = comparator;
                this.batchSize = batchSize;
                this.currentPosition = new HashMap<>();
                this.values = new HashMap<>();
                for ( String ident : queryMap.keySet() ) {
```

```
                        this.values.put( ident, new LinkedList<>() );
                        this.currentPosition.put( ident, 0L );
                }
        }

        private static int toInt(Long l) {
                return (int) (long) l;
        }

        /**
         * increments the value to be returned by {@link #get()}
         *
         * @return true if there is a value left to be visited
         *        in the database
         */
        public boolean next() {

        /*
         *
         *
         * indentation broken to make this readable
         *
         *
         */

this.scheduled = null;
this.identifier = null;
List<ObjectIdentifierWrapper> tmp =
        new ArrayList<>( this.queryMap.size() );

for ( Map.Entry<String, Query> entry : this.queryMap.entrySet() ) {
        String identifier = entry.getKey();
        Query query = entry.getValue();
        if ( !this.currentCountMap.get( identifier ).equals( 0L ) ) {
                if ( this.values.get( identifier ).size() == 0 ) {
                        // the last batch is empty. get a new one
                        Long processed =
                                this.currentPosition.get( identifier );
                        // yay JPA...
                        query.setFirstResult( toInt( processed ) );
                        query.setMaxResults( this.batchSize );
                        @SuppressWarnings("unchecked")
                        List<Object> list = query.getResultList();
                        this.values.get( identifier ).addAll( list );
                }
                Object val = this.values.get( identifier ).getFirst();
                tmp.add( new ObjectIdentifierWrapper( val, identifier ) )
                    ;
```

```
	}
}
tmp.sort( this.comparator );
if ( tmp.size() > 0 ) {
	ObjectIdentifierWrapper arr = tmp.get( 0 );
	this.scheduled = arr.object;
	this.identifier = arr.identifier;
	this.values.get( this.identifier ).pop();
	Long currentPosition = this.currentPosition.get( arr.identifier );
	Long newCurrentPosition =
		this.currentPosition
			.computeIfPresent( arr.identifier,
				(clazz, old) -> old + 1 );
	if ( Math.abs( newCurrentPosition - currentPosition ) != 1L ) {
		throw new AssertionFailure(
			"the new currentPosition count " +
			"should be exactly 1 " +
			"greater than the old one" );
	}
	Long count = this.currentCountMap.get( arr.identifier );
	Long newCount = this.currentCountMap.computeIfPresent(
		arr.identifier, (clazz, old) -> old - 1
	);
	if ( Math.abs( count - newCount ) != 1L ) {
		throw new AssertionFailure(
			"the new old remaining count " +
				should be exactly 1 " +
				"greater than the new one" );
	}
}
return this.scheduled != null;
}

	/**
	* @return the current value
	*/
	public Object get() {
		if ( this.scheduled == null ) {
			throw new IllegalStateException(
				"either empty or next() has " +
					"not been called" );
		}
		return this.scheduled;
	}

	/**
	* @return the identifier of the current value
```

```
        */
        public String identifier() {
                if ( this.identifier == null ) {
                        throw new IllegalStateException(
                                "either empty or next() has " +
                                        "not been called" );
                }
                return this.identifier;
        }

        public static class ObjectIdentifierWrapper {

                public final Object object;
                public final String identifier;

                public ObjectIdentifierWrapper(Object object,
                        String identifier) {
                        this.object = object;
                        this.identifier = identifier;
                }

        }

}
```

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] Web Engineering, chapter: Requirements Engineering: Problem Analysis and Solution Specification, R. J. Wieringa, 2004

[12] objectmentor.com: Article on Single Responsibility Principle http://www.objectmentor.com/resources/articles/srp.pdf, 09/04/2015

[13] objectmentor.com: Article on Open-Closed-Principle http://www.objectmentor.com/resources/articles/ocp.pdf, 09/04/2015

[14] Object-Oriented Software Construction, Prentice Hall, 1988, Bertrand Meyer

[15] Maven project homepage https://maven.apache.org/, 09/04/2015

[16] Top-down programming, Robert Strandh http://dept-info.labri.fr/~strandh/Teaching/PFS/Common/Strandh-Tutorial/top-down-programming.html, 09/04/2015

[17] Bottom-up programming, Robert Strandh http://dept-info.labri.fr/~strandh/Teaching/PFS/Common/Strandh-Tutorial/bottom-up-programming.html, 09/04/2015

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

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

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

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

[22] Stackoverflow JPA tag http://stackoverflow.com/tags/jpa/info, 09/04/2015

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

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

[25] JSR 338: JPA 2.1 specification https://jcp.org/en/jsr/detail?id=338, 08/27/2015

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

[27] Java EE specification on oracle.com http://www.oracle.com/technetwork/java/javaee/tech/index.html, 09/04/2015

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

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

[30] ElasticSearch Homepage https://www.elastic.co/products/elasticsearch, 08/19/2015

[31] Solr Homepage http://lucene.apache.org/solr/, 08/19/2015

[32] ElasticSearch Download website https://www.elastic.co/downloads/elasticsearch, 08/27/2015

[33] Solr security https://wiki.apache.org/solr/SolrSecurity, 08/19/2015

[34] elastic Shield (security for ElasticSearch) https://www.elastic.co/products/shield, 08/19/2015

[35] Solr Administration (Core Specific Tools) https://cwiki.apache.org/confluence/display/solr/Core-Specific+Tools, 08/19/2015

[36] ElasticHQ http://www.elastichq.org/, 08/19/2015

[37] ElasticSearch: Life inside a cluster https://www.elastic.co/guide/en/elasticsearch/guide/current/distributed-cluster.html, 08/19/2015

[38] Solr: Introduction to Scaling and Distribution https://cwiki.apache.org/confluence/display/solr/Introduction+to+Scaling+and+Distribution, 08/19/2015

[39] hibernate-search-engine on mvnrepository.org http://mvnrepository.com/artifact/org.hibernate/hibernate-search-engine/5.3.0.Final, 08/27/2015

[40] Hibernate Search documentation http://hibernate.org/search/documentation/, 07/31/2015

[41] 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

[42] MySQL - Developer's Library, Fourth Edition, 2009, Paul DuBois

[43] CREATE TRIGGER in PostgreSQL http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html, 08/30/2015

[44] Triggers in HSQLDB http://hsqldb.org/doc/guide/triggers-chapt.html, 08/30/2015

[45] Triggers in Firebird http://www.firebirdsql.org/refdocs/langrefupd21-ddl-trigger.html, 08/30/2015

[46] Truncate statement PostgreSQL docs http://www.postgresql.org/docs/9.1/static/sql-truncate.html, 08/27/2015

[47] Firebird conformance http://www.firebirdsql.org/en/sql-conformance/, 08/27/2015

[48] Hibernate Search GenericJPA GitHub repository https://github.com/Hotware/Hibernate-Search-GenericJPA, 08/13/2015

[49] Hibernate Search roadmap http://hibernate.org/search/roadmap/, 08/14/2015

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

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.

$Unterschrift:$ $Ort, Datum:$