UNIVERSITÄT
BAYREUTH

University of Bayreuth
Department of Informatics

Bachelor Thesis

in Informatics

Topic:	Integration of JPA-conform ORM-Implementations in Hibernate Search
Author:	Martin Braun <martinbraun123@aol.com> Matrikel-Nr. 1249080
Version date:	August 19, 2015
1. Supervisor:	Dr. Bernhard Volz
2. Supervisor:	Prof. Dr. Bernhard Westfechtel

Zusammenfassung

Abstract

Contents

1 Preface

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

The Java Persistence API (JPA) went even further by standardising ORMs. First conceived in 2006 [1], it is now the de-facto standard for Object Relational Mappers in Java. The developer doesn't need to know which specific ORM is used in the application, as all the database queries arewritten against a standardized query API and therefore portable. This means that not only the database is interchangeable, but even the specific ORM, it is accessed by, is as well.

However, this does not mean that all JPA implementations come with the same features. While all of them are JPA compliant (apart from minor bugs), some ship with additional modules to enhance their capabilities. A perfect example for this is the Hibernate Search API aimed at Hibernate ORM users.[2] [3]

Nowadays, even small applications like online shops need enhanced search capabilities to let the user find more results for a given input. This is not something a regular RDBMS excels at and Hibernate Search comes into use: It works atop the Hibernate ORM/JPA system and enables the developer to index the domain model for searching. It's not only a mapper from JPA entities to a search index, but also keeps the index up-to-date if something in the database changes.

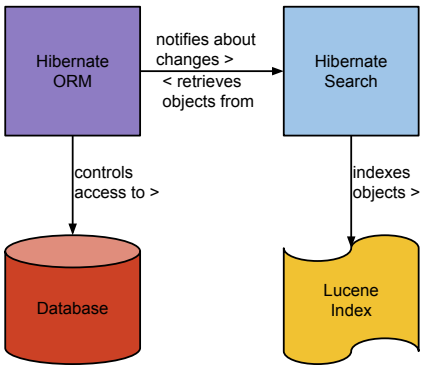

Figure 1: Hibernate Search with Hibernate ORM

[1] Wikipedia on Java Persistence API, see [1]
[2] Hibernate ORM project homepage, see [11]
[3] Hibernate Search project homepage, see [2]

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

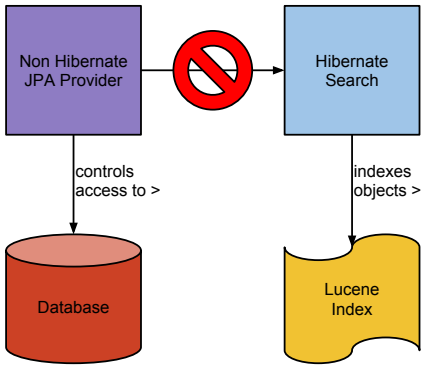

Figure 2: Hibernate Search's incompatibility with other JPA implementations

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

Currently, developers that do not want to bind themselves to Hibernate ORM have to resort to using different full text search systems like native Lucene[4], ElasticSearch[5] or Solr[6]. While this is always a viable option, for some applications Hibernate Search would be a much better suit because of it's design with a entity structure in mind and the automatic index updating feature, if it just were compatible with generic JPA.

When investigating Hibernate Search's project structure [7], we can see that the only module apart from some server-integration modules that depends on any ORM logic is "hibernate-search-orm". The modules that contain the indexing engine, the replication logic, alternative backends, etc. are completely independent from any ORM logic. This means, that most of the codebase could be reused for a generic version of Hibernate Search.

[4]official Lucene website, see [20]
[5]ElasticSearch Java API, see [6]
[6]Solr Java API, see [7]
[7]Hibernate Search GitHub repository, see [15]

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

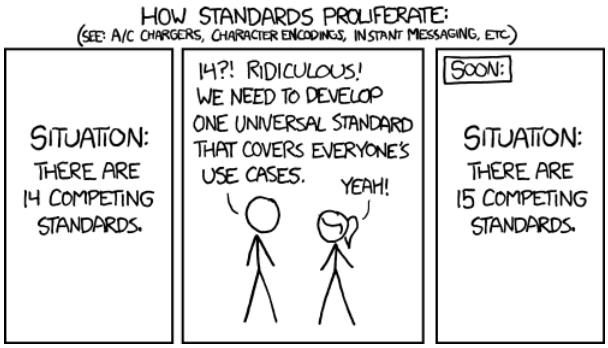

Figure 3: xkcd.com on competing standards [8]

This is why 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.

[8] xkcd comic #927, see [18]

2 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 relevant technologies first. We will explain why ORMs in general and the JPA specification in particular are beneficial. Then, we will explain what fulltext search engines are used for and give a short overview about the available solutions for Java. We will see that generalizing Hibernate Search for any JPA implementation is a good approach and that it has benefits over using the different search solutions available.

2.1 Object Relational Mappers

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

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

Listing 1: SQL "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.

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

Listing 2: ORM query example

[9] Wikipedia on Object Oriented Programming (OOP), see [8]
[10] Oracle JDBC overview, see [16]
[11] OleDb usage page, see [17]

This is especially useful if used in big software products as not all programmers have to know the exact details of the underlying database. The database system could even be completely replaced for another (provided the ORM supports the specific RDBMS), while the business logic would not changing a bit.

2.2 JPA

The first version of the JPA standard was released in May 2006. From then on it rose to being probably the most commonly used persistence API for Java and is considered the "industry standard approach for Object Relational Mapping"[12]. While mostly known for standardizing relational database mappers (ORM), it also supports other concepts like NoSQL[13] [14] or XML storage[15]. However, when talking about JPA in this thesis, we will be focusing on the relational aspects of it. Currently, the newest version of this standard is 2.1.[16].

Some popular relational implementations are:

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

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

This is particularily important if you are working in a Java EE environment. Java EE itself is a specification for platforms, mostly Web-servers (JPA is part of the Java EE spec).[20] Many Java EE Web-servers ship with a bundled JPA implementation that they are optimized for (Wildfly with Hibernate ORM, GlassFish with EclipseLink, ...). This means that if the server is switched, it could also be a reasonable idea to swap out the JPA implementor. If everything in the application is written in a JPA compliant way, the user will then generally not run into many problems related to this switch.

[12]Wikibooks on Java Persistence, see [9]
[13]Hibernate OGM project homepage, see [10]
[14]EclipseLink project homepage, see [14]
[15]EclipseLink project homepage, see [14]
[16]Wikipedia on Java Persistence API, see [1]
[17]Hibernate ORM project homepage, see [11]
[18]EclipseLink project homepage, see [14]
[19]OpenJPA project homepage, see [12]
[20]Wikipedia on Java EE, see [19]

2.3 Fulltext search engines

Conventional relational databases are good at retrieving and querying structured data. But if one wants to build a search engine atop a domain model, most RDBMS will only support the SQL-LIKE operator [21]:

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

Listing 3: SQL LIKE operator in use

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

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

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

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

[21] w3schools on SQL LIKE, see [13]

2.3.1 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.[22]

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

2.3.1.1 Concepts As Lucene's focus is not on storing relational data, it comes with its own set of concepts. Following is a short overview over the 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.

2.3.1.1.1 Index structure Lucene uses an **inverted index** to store data. This means that instead of storing texts mapped to the words contained in them, it works the other way around. All different words (terms) are mapped to the texts they occur in[23], so it can be compared to a $Map < String, List < Text >>$ in Java. Before anything can be searched using Lucene, it has to be added to the the index (indexed) first.

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

2.3.1.1.3 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).

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

[22] official Lucene website, see [20]
[23] Lucene basic concepts, see [21]

2.3.1.2 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.

2.3.1.3 Features Lucene probably is the most complete toolbox to build a search-engine from. It has pre-built analyzers for many languages, a queryparser to support 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.

2.3.2 Fulltext search servers: ElasticSearch and Solr

Lucene is the basis for two of the most popular search servers available: ElasticSearch (by elastic)[24] and Solr (sister project of Lucene)[25].

2.3.2.1 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 [26] [27]. Index changes and queries are done via a REST-like API (among other options).

2.3.2.2 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, ... [28] [29]). They are generally used because of their good clustering capabilities (distribution & replication) and are optimized for high throughput and scalability [30] [31]. 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 this case Java).

[24] ElasticSearch Homepage, see [30]
[25] Solr Homepage, see [31]
[26] Solr security, see [24]
[27] elastic Shield (security for ElasticSearch), see [25]
[28] Solr Administration (Core Specific Tools), see [26]
[29] ElasticHQ, see [27]
[30] ElasticSearch: Life inside a cluster, see [28]
[31] Solr: Introduction to Scaling and Distribution, see [29]

2.3.3 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. [32]

2.3.3.1 Usage Hibernate Search is used in the context of JPA compliant applications using Hibernate ORM. It 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.

2.3.3.2 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 its optional Infinispan integration.

[32] Hibernate Search GitHub README, see [15]

2.3.4 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. Python), 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 [33]).

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.

[33] see `http://mvnrepository.com/artifact/org.compass-project/compass/2.2.0`

3 Challenges

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

3.1 The example project

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

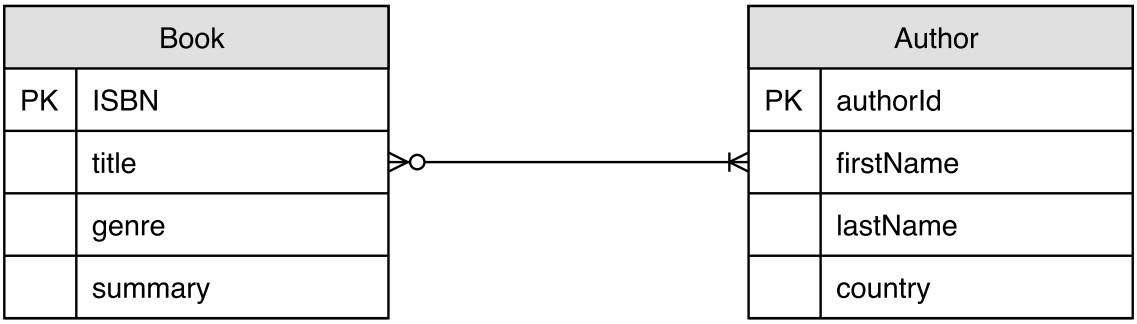

Figure 4: the bookstore entity relationship model

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

3.2 Indexing & searching

Hibernate Search's engine wasn't designed to be used directly by application developers. Its main purpose is to serve as an integration point for other APIs that need to leverage its power to index object graphs and query the index for hits. This is why we have to write our own standalone module based on the "hibernate-search-engine" to ease its general usage. After the standalone is finished, we will build an integration of it with JPA to have a user experience as similar to Hibernate Search ORM as possible. 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.

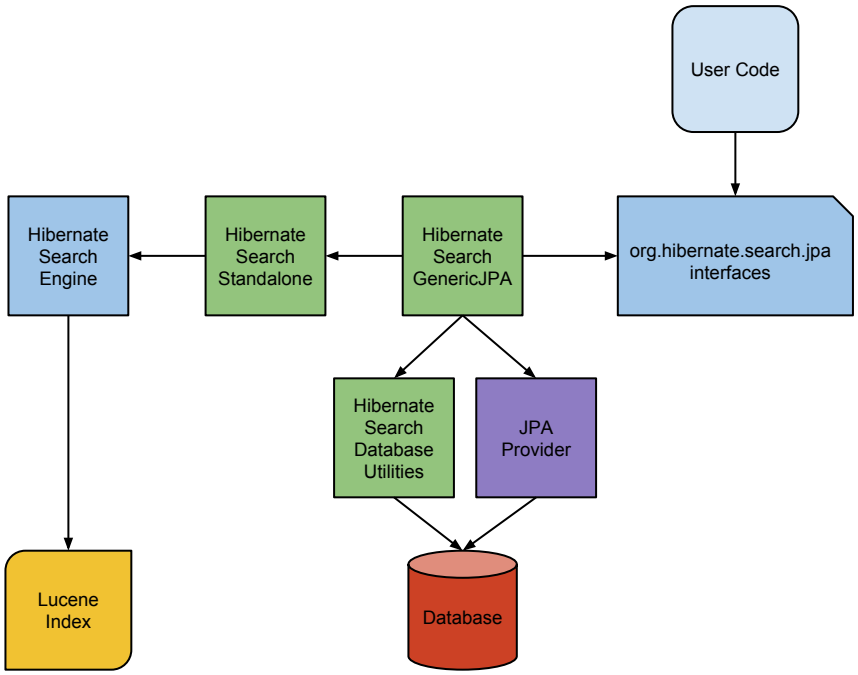

Figure 5: Complete Architecture of Hibernate Search GenericJPA

3.3 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.

4 Building a JPA integration on top of Hibernate Search

In this section we will start by discussing how Hibernate Search's engine (in the form of the module "hibernate-search-engine") can be used in general. Then we will work out a standalone version of this engine that is easier to work with and lastly we will show how we integrate this standalone version with JPA.

4.1 Setting up the example project

Before we explain how we do things in particular, we set up the example entities described in 3.1 as if the original Hibernate Search would have been used. We do so by adding additional annotations to our entity-classes:

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.
3. **@Field**: describes how the annotated field should be indexed. The fieldname defaults to the Java property name.
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.
5. **@ContainedIn**: used in entities that are embedded in other indexes. this is set on the properties that point back to the index-owning entity.

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

The resulting entities look like this:

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

Listing 6: Book.java with Hibernate Search annotations

```
@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

4.2 Using Hibernate Search's engine

As already described earlier (3.2), 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 and all following information had to be retrieved from tests in the hibernate-search-engine and hibernate-search-orm integration module source code.

4.2.1 Starting the engine

A Hibernate Search engine instance is represented by a **SearchIntegrator**. 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 is "4.10.4".

A complete list of the available settings can be found in the Hibernate Search documentation [34] (only the Hibernate ORM specific settings cannot be used). Our **StandaloneSearchConfiguration** (appendix listing 40) defaults to "ram" and "4.10.4".

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

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

[34] Hibernate Search documentation, see [3]

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

Listing 8: Starting up the engine

4.2.2 Indexing, updating and deleting objects from the index

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 39). The different index operations are represented by **Work** objects that contain the WorkType (INDEX, UPDATE, PURGE, etc.) and all necessary data to execute the individual task.

Indexing objects with **WorkType.INDEX**:

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

Listing 9: Indexing an object with the engine

Updating objects with **WorkType.UPDATE**:

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

Listing 10: Updating an object with the engine

Deleting objects with **WorkType.PURGE**:

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

Listing 11: Deleting an object by id with the engine

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

4.2.3 Querying the index

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).

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

Listing 12: Querying the index with the engine

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.

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

Listing 13: Extracting info from the results

4.3 Standalone version of Hibernate Search

In 4.2 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 feasible 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. Note that we are using a specialization of **SearchIntegrator** - namely **ExtendedSearchIntegrator** - which allows us to have more sophisticated features.

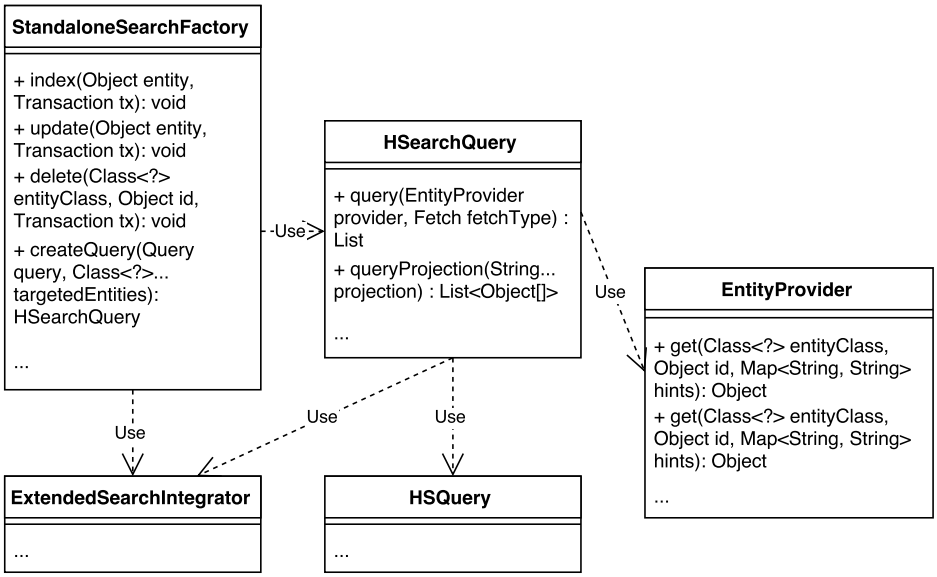

Figure 6: Rough architecture of the standalone (important parts)

4.3.1 Starting the standalone

The startup process of the standalone 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 an the actual implementation object.

```
1  List<Class<?>> indexClasses = Arrays.asList(Book.class, Author.class);
2
3  SearchConfiguration searchConfiguration =
4                  new StandaloneSearchConfiguration();
5  indexClasses.forEach( searchConfiguration::addClass );
6
7  StandaloneSearchFactory searchFactory =
8                  StandaloneSearchFactoryFactory.
9                                  createSearchFactory(
10                                         searchConfiguration,
11                                         indexClasses
12                                 );
13
14 //use the searchfactory ...
15
16 //close it
17 searchFactory.close();
```

Listing 14: Starting up the standalone

4.3.2 Indexing, updating and deleting objects from the index

With our standalone version, basic index control becomes more streamlined as we don't have to work with SearchIntegrator's Worker pattern anymore.

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

Listing 15: Indexing an object with the standalone

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

Listing 16: Updating an object with the standalone

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

Listing 17: Deleting an object by id with the standalone

4.3.3 Querying the index

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** (41). Besides taking a 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 (42). After its creation this map can then be stored in a central place and reused.

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

Listing 18: Querying the index with the standalone

4.4 Standalone integration with JPA interfaces

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 - 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.

4.4.1 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 only contains only the methods relevant for the following sections.

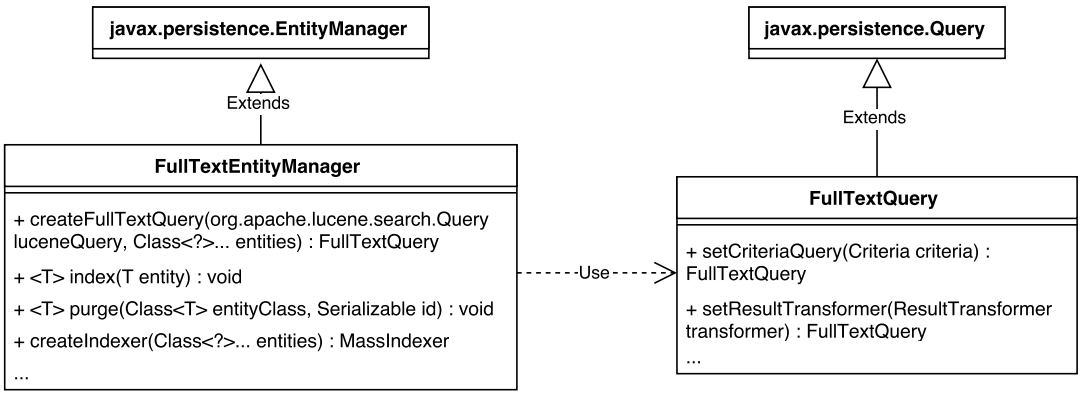

Figure 7: The main JPA interfaces of Hibernate Search ORM

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

```
1 ...
2 <property name="hibernate.search.default.directory_provider"
3 value="filesystem"/>
4 <property name="hibernate.search.default.indexBase"
5 value="/path/to/indexes"/>
6 ...
```

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

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

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

Listing 20: Obtaining a FullTextEntityManager with Hibernate Search ORM

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

4.4.1.2 Indexing, updating and deleting objects from the index

The index operations are all straightforward and similar to what we designed our Standalone integration in 4.3 to work like apart from minor naming differences.

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

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

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

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

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

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

4.4.1.3 Querying the index Hibernate Search ORM integrates even better with JPA for queries than our Standalone version as the FullTextQuery interfaces extends the JPA Query interface and uses getResultList() to return its results.

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

Listing 23: Querying with Hibernate Search ORM

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

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

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

Listing 24: MassIndexer usage with Hibernate Search ORM

"This will rebuild the index of all [Book] instances (and subtypes), and will create 12 parallel threads to load the User instances using batches of 25 objects per query; these same 12 threads will also need to process indexed embedded relations and custom FieldBridges or ClassBridges, to finally output a Lucene document."[35]

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

4.4.2 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 all together.

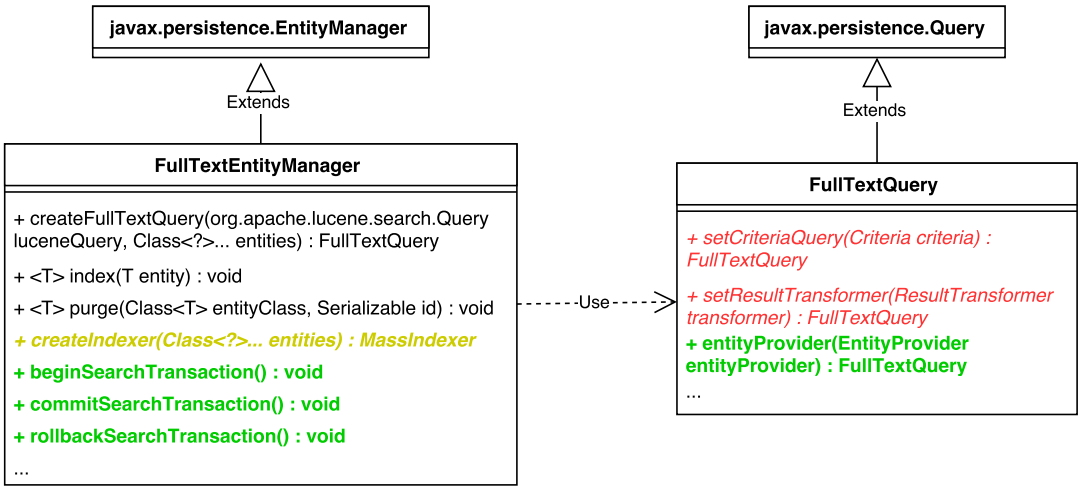

Figure 8: Required fixes for a generic version

In the figure above, we marked all the methods needing fixing in the FullTextEntity-Manager and FullTextQuery interfaces. Besides these, some other aspects need changing as well. We will discuss all of the needed changes & additions in the following paragraphs.

4.4.2.1 Starting 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 **JPASearchFactoryController**.

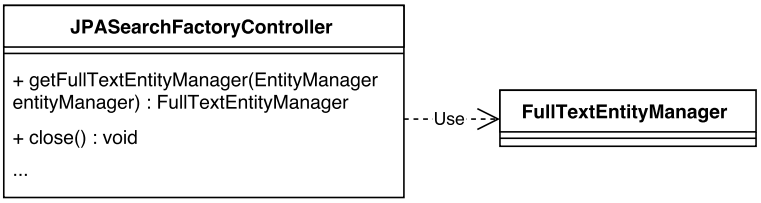

Figure 9: JPASearchFactoryController

Having this separate interface means that its lifecycle has to be controlled on its own. We start it with our bootstrapping class **Setup** like this:

```
1  EntityManagerFactory emf = ...;
2  Properties properties = new Properties();
3
4  properties.setProperty(
5          "hibernate.search.searchfactory.type",
6          "manual-updates"
7  );
8
9  JPASearchFactoryController searchFactoryController =
10         Setup.createSearchFactoryController( emf, properties );
11
12 //use it...
13
14 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.

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. This means that an instance of the JPASearchFactoryController has to be available at all times 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 we want to index POJOs which are not associated with JPA (see table 2 for the property to work with these additional entities).

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

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

Listing 26: Index control with Hibernate Search GenericJPA

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.

```
1  // get the first class in the hierarchy that is actually in the index
2  Class<T> clazz = (Class<T>) entity.getClass();
3  while ( (clazz = (Class<T>) clazz.getSuperclass()) != null ) {
4          if ( clazz.isAnnotationPresent( InIndex.class ) ) {
5                  break;
6          }
7  }
8  if ( clazz != null ) {
9          return clazz;
10 }
11 //no @InIndex found, try this class
12 return entity.getClass();
```

Listing 27: Algorithm to determine the actual indexed type

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

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

Listing 28: Book.java with @InIndex

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

Listing 29: Author.java with @InIndex

4.4.2.3 Querying the index While we didn't mention this in 4.4.1.3, Hibernate Search ORM supports modifying the resulting objects of a query with these two methods:

- **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 4.3.3 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.

4.4.2.4 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 a 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:

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

Listing 30: MassIndexer usage with Hibernate Search ORM

4.5 The automatic index updating feature

As already stated in 3.3, 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 compared to the short introductions of the other features.

4.5.1 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 at every point in time index 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.

4.5.1.1 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.

4.5.1.1.1 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:

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

Listing 31: Example JPA entity listener

It is then applied with an annotation on the entity:

```
1  @EntityListeners( { EntityListener.class } )
2  public class Book {
3
```

```
4         //...
5
6 }
```

Listing 32: Using a JPA entity listener

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

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

Listing 33: Injecting the EntityListener

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

Firstly, not all JPA providers seem to handle these events similar: 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.

```
1 EntityManager em = ...;
2
3 em.getTransaction().begin();
4
5 Book book = em.find( Book.class "someIsbn" );
6 book.setTitle( "someNewTitle" );
7
8 // flushes, so we retrieve the Book with the changes from above
9 // => event is triggered
10 List<Book> allBooks =
11         em.createQuery( "SELECT b FROM Book b" ).getResultList();
```

```
12
13 // we have no way to get this event to revert the wrong index change
14 em.getTransaction().rollback();
```

Listing 34: Event triggering on flush

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

4.5.1.1.2 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.

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.

4.5.1.2 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.

Triggers are "procedural code that is automatically executed in response to certain events on a particular table or view in a database" [36]. While they are mostly "used for maintaining the integrity of the information on the database" [37], they are also useful for listening to events.

Almost every RDBMS at least supports triggers on the three crucial events for event-listening: INSERT (CREATE), UPDATE, DELETE.

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 we don't need, but still keep the index up-to-date.

[36] Wikipedia on RDBMS triggers, see [5]
[37] Wikipedia on RDBMS triggers, see [5]

4.5.1.2.1 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 shows the trigger architecture needed for our Author and Book example.

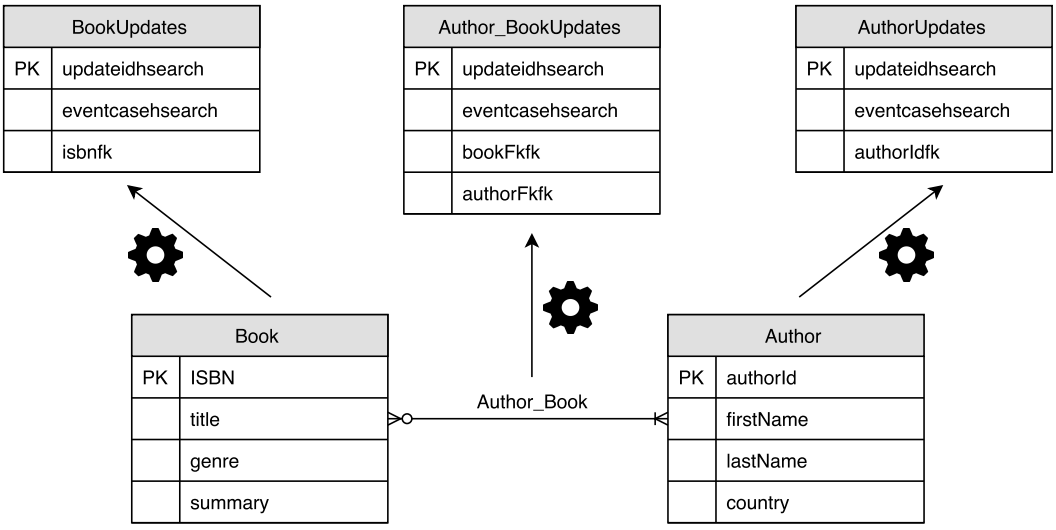

Figure 10: Triggers for the example project

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 then (we can't have a reference to a non existent entity).

4.5.1.2.2 Creating the tables Since the creation of these tables requires a lot of work to be done, we have to automate it as good as possible. We do this by requiring additional 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 and the types & names of the entity key columns. The name of the update table and the columns in it is then generally derived automatically from that.

Note: The update tables are NO JPA entities, so we have to work with native SQL in the backend

```
@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

```
@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

However, if the developer needs different names in the update tables, 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 using these.

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.

4.5.1.2.3 Retrieving the events 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 is doing this for the table author_bookupdates with standard SQL that should be working on every RDBMS.

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

Listing 37: Querying for updates (Author_Book)

We run queries of this type for every update table with fixed delays (configurable, see 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 44 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 (A 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.

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

Listing 38: Deleting handled updates (Author_Book)

With these two types of queries for each update table we are able to keep the index up-to-date efficiently and also make sure that no event is handled twice.

4.5.2 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.

4.5.2.1 Additional work Since the native event system gets the proper information about changes from the vendor side, it doesn't require a lot information about the general structure of the domain model and tables in the database. For the Trigger based event system, that's a different story as it has to poll info about changes from the database. This is the reason the user has to add this information as we have seen in 4.5.1.2.2.

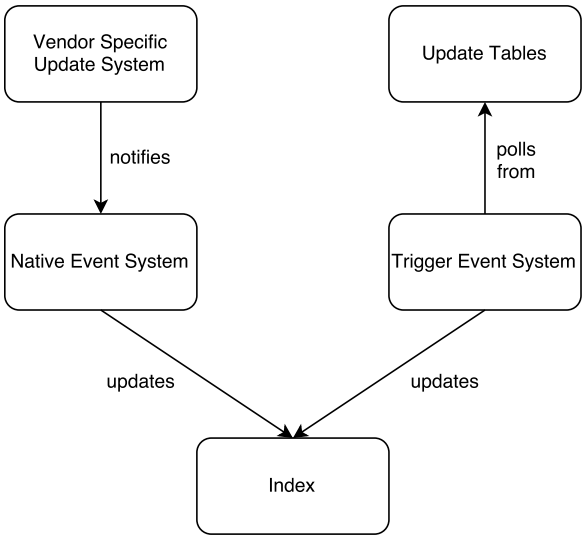

Figure 11: Hibernate Search GenericJPA update mechanisms

4.5.2.2 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).

4.5.2.3 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

5 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 3 have been resolved.

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 reason is that the ultimate goal for this project is to be merged into the original Hibernate Search codebase.

This is the reason this project has to be looked as a proof of concept even though the code as it can be found on GitHub [38] can already be used in real applications.

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 early October 2015. This comes exactly at the right moment as the Hibernate Search team is planning API changes in the near future [39] as some interfaces have to be altered as we have seen in 4.4.

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 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. And competition has never been a bad thing.

[38] Hibernate Search GenericJPA GitHub repository, see [22]
[39] Hibernate Search roadmap, see [23]

References

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

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

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

[4] 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

[5] Wikipedia on RDBMS triggers https://en.wikipedia.org/wiki/Database_trigger, 08/12/2015

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listings

```
1  public class Transaction implements TransactionContext {
2
3          private boolean progress = true;
4          private List<Synchronization> syncs = new ArrayList<>();
5
6          @Override
7          public boolean isTransactionInProgress() {
8                  return this.progress;
9          }
10
11         @Override
12         public Object getTransactionIdentifier() {
13                 return this;
14         }
15
16         @Override
17         public void registerSynchronization(
18                 Synchronization synchronization ) {
19                 this.syncs.add( synchronization );
20         }
21
22         /**
23          * @throws IllegalStateException if already commited/rolledback
24          */
25         public void commit() {
26                 if ( !this.progress ) {
27                         throw new IllegalStateException(
28                         "can't commit - " +
29                         "No Search Transaction is in Progress!" );
30                 }
31                 this.progress = false;
32                 this.syncs.forEach( Synchronization::beforeCompletion );
33
34                 for ( Synchronization sync : this.syncs ) {
35                         sync.afterCompletion( Status.STATUS_COMMITTED );
36                 }
37         }
38
39         /**
40          * @throws IllegalStateException if already commited/rolledback
41          */
42         public void rollback() {
43                 if ( !this.progress ) {
44                         throw new IllegalStateException(
45                         "can't rollback - " +
46                         "No Search Transaction is in Progress!" );
```

```
47                  }
48                  this.progress = false;
49                  this.syncs.forEach( Synchronization::beforeCompletion );
50
51                  for ( Synchronization sync : this.syncs ) {
52                          sync.afterCompletion( Status.STATUS_ROLLEDBACK );
53                  }
54          }
55
56  }
```

Listing 39: the simple Transaction contract

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

```
37          }
38
39          public StandaloneSearchConfiguration(InstanceInitializer init) {
40                  this( new Properties() );
41          }
42
43          public StandaloneSearchConfiguration(InstanceInitializer init,
44                  Properties properties) {
45                  this.initializer = init;
46                  this.classes = new HashMap<>();
47                  this.properties = properties;
48                  // default values if nothing was explicitly set
49                  this.properties.computeIfAbsent(
50                          "hibernate.search.default.directory_provider",
51                          (key) -> {
52                                  LOGGER.info(
53                                    "defaulting to RAM directory-provider"
54                                  );
55                          return "ram";
56                  });
57                  this.properties.computeIfAbsent(
58                          "hibernate.search.lucene_version",
59                          (key) -> {
60                                  LOGGER.info(
61                                          "defaulting to Lucene Version: "
62                                          + Version.LUCENE_4_10_4.toString()
63                                  );
64                                  return Version.LUCENE_4_10_4.toString();
65                  });
66                  this.reflectionManager = new JavaReflectionManager();
67                  this.providedServices = new HashMap<>();
68                  this.classLoaderService = new DefaultClassLoaderService();
69          }
70
71          public StandaloneSearchConfiguration addProperty(String key,
72                  String value) {
73                  properties.setProperty( key, value );
74                  return this;
75          }
76
77          public StandaloneSearchConfiguration addClass(Class<?> indexed) {
78                  classes.put( indexed.getName(), indexed );
79                  return this;
80          }
81
82          @Override
83          public Iterator<Class<?>> getClassMappings() {
84                  return classes.values().iterator();
```

```
85          }
86
87          @Override
88          public Class<?> getClassMapping(String name) {
89                  return classes.get( name );
90          }
91
92          @Override
93          public String getProperty(String propertyName) {
94                  return properties.getProperty( propertyName );
95          }
96
97          @Override
98          public Properties getProperties() {
99                  return properties;
100         }
101
102         @Override
103         public ReflectionManager getReflectionManager() {
104                 return this.reflectionManager;
105         }
106
107         @Override
108         public SearchMapping getProgrammaticMapping() {
109                 return programmaticMapping;
110         }
111
112         public StandaloneSearchConfiguration setProgrammaticMapping(
113                         SearchMapping programmaticMapping
114                 ) {
115                 this.programmaticMapping = programmaticMapping;
116                 return this;
117         }
118
119         @Override
120         public Map<Class<? extends Service>, Object>
121                 getProvidedServices() {
122                 return providedServices;
123         }
124
125         public void addProvidedService(
126                         Class<? extends Service> serviceRole,
127                         Object service
128                 ) {
129                 providedServices.put( serviceRole, service );
130         }
131
132         @Override
```

```
133         public boolean isTransactionManagerExpected() {
134                 return this.transactionsExpected;
135         }
136
137         public void setTransactionsExpected(
138                         boolean transactionsExpected) {
139                 this.transactionsExpected = transactionsExpected;
140         }
141
142         @Override
143         public InstanceInitializer getInstanceInitializer() {
144                 return initializer;
145         }
146
147         @Override
148         public boolean isIndexMetadataComplete() {
149                 return indexMetadataComplete;
150         }
151
152         public void setIndexMetadataComplete(
153                 boolean indexMetadataComplete) {
154                 this.indexMetadataComplete = indexMetadataComplete;
155         }
156
157         @Override
158         public boolean isIdProvidedImplicit() {
159                 return idProvidedImplicit;
160         }
161
162         public StandaloneSearchConfiguration
163                 setIdProvidedImplicit(boolean idProvidedImplicit) {
164                 this.idProvidedImplicit = idProvidedImplicit;
165                 return this;
166         }
167
168         @Override
169         public ClassLoaderService getClassLoaderService() {
170                 return classLoaderService;
171         }
172
173         public void setClassLoaderService(
174                 ClassLoaderService ) {
175                 this.classLoaderService = classLoaderService;
176         }
177
178 }
```

Listing 40: StandaloneSearchConfiguration.java

```
1  public class BasicEntityProvider implements EntityProvider {
2
3          private static final String QUERY_FORMAT =
4                  "SELECT obj FROM %s obj " +
5                  "WHERE obj.%s IN :ids";
6          private final EntityManager em;
7          private final Map<Class<?>, String> idProperties;
8
9          public BasicEntityProvider(EntityManager em,
10                 Map<Class<?>, String> idProperties) {
11                 this.em = em;
12                 this.idProperties = idProperties;
13         }
14
15         @Override
16         public void close() throws IOException {
17                 this.em.close();
18         }
19
20         @Override
21         public Object get(Class<?> entityClass, Object id,
22                 Map<String, String> hints) {
23                 return this.em.find( entityClass, id );
24         }
25
26         @SuppressWarnings({"rawtypes", "unchecked"})
27         @Override
28         public List getBatch(Class<?> entityClass, List<Object> ids,
29                 Map<String, String> hints) {
30                 List<Object> ret = new ArrayList<>( ids.size() );
31                 if ( ids.size() > 0 ) {
32                         String idProperty =
33                                 this.idProperties.get( entityClass );
34                         String queryString =
35                                 String.format(
36                                         QUERY_FORMAT,
37                                         this.em.getMetamodel()
38                                                 .entity( entityClass )
39                                                 .getName(),
40                                         idProperty
41                 );
42                 Query query = this.em.createQuery( queryString );
43                 query.setParameter( "ids", ids );
44                         ret.addAll( query.getResultList() );
45                 }
46                 return ret;
47         }
```

```
48
49          public void clearEm() {
50                  this.em.clear();
51          }
52
53          public EntityManager getEm() {
54                  return this.em;
55          }
56
57  }
```

Listing 41: BasicEntityProvider.java

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

Listing 42: Obtaining idProperties

```
1  /**
2   * @author Emmanuel Bernard
3   * @author Martin Braun
4   */
5  public interface FullTextEntityManager extends EntityManager {
6
7          /**
8           * Create a fulltext query on top of a native Lucene
9           * query returning the matching objects of columnTypes
10          * <code>entities</code> and their respective subclasses.
11          *
12          * @param luceneQuery The native Lucene query to be
13          *                    run against the Lucene index.
14          * @param entities List of classes for columnTypes filtering.
15          *                    The query result will only return entities
```

```
16          *               of the specified types and their respective
17          *               subtype.
18          *               If no class is specified no columnTypes filtering
19          *               will take place.
20          *
21          * @return A <code>FullTextQuery</code> wrapping around the
22          *       native Lucene query.
23          *
24          * @throws IllegalArgumentException if entityType is
25          * <code>null</code> or not a class or superclass annotated with
26          * <code>@Indexed</code>.
27          */
28         FullTextQuery createFullTextQuery(
29                 org.apache.lucene.search.Query luceneQuery,
30                 Class<?>... entities);
31
32         /**
33          * Force the (re)indexing of a given <b>managed</b> object.
34          * Indexation is batched per search-transaction: if a
35          * transaction is active, the operation will not affect
36          * the index at least until commit.
37          *
38          * @param entity The entity to index
39          *       - must not be <code>null</code>.
40          *
41          * @throws IllegalArgumentException
42          *       if entity is null or not an @Indexed entity
43          * @throws IllegalStateException
44          *       if no search-transaction is in progress
45          */
46         <T> void index(T entity);
47
48         /**
49          * @return the <code>SearchFactory</code> instance.
50          */
51         SearchFactory getSearchFactory();
52
53         /**
54          * Remove the entity with the columnTypes
55          * <code>entityType</code> and the identifier
56          * <code>id</code> from the index. If
57          * <code>id == null</code> all indexed entities
58          * of this columnTypes and its indexed subclasses
59          * are deleted. In this case
60          * this method behaves like {@link #purgeAll(Class)}.
61          *
62          * @param entityType The columnTypes of the
63          *       entity to delete.
```

```
64          * @param id The id of the entity to delete.
65          *
66          * @throws IllegalArgumentException if entityType is
67          * <code>null</code> or not a class or superclass
68          * annotated with <code>@Indexed</code>.
69          * @throws IllegalStateException if no
70          * search-transaction is in progress
71          */
72         <T> void purge(Class<T> entityType, Serializable id);
73
74         /**
75          * Remove all entities from of particular class
76          * and all its subclasses from the  index.
77          *
78          * @param entityType The class of the entities to remove.
79          *
80          * @throws IllegalArgumentException if entityType is
81          *       <code>null</code> or not a class or superclass
82          * annotated with <code>@Indexed</code>.
83          * @throws IllegalStateException if no search-transaction
84          * is in progress
85          */
86         <T> void purgeAll(Class<T> entityType);
87
88         <T> void purgeByTerm(Class<T> entityType,
89                 String field,
90                 Integer val);
91
92         <T> void purgeByTerm(Class<T> entityType,
93                 String field,
94                 Long val);
95
96         <T> void purgeByTerm(Class<T> entityType,
97                 String field,
98                 Float val);
99
100        <T> void purgeByTerm(Class<T> entityType,
101                String field,
102                Double val);
103
104        <T> void purgeByTerm(Class<T> entityType,
105                String field,
106                String val);
107
108        /**
109         * Flush all index changes forcing Hibernate Search to apply all
110         * changes to the index not waiting for the batch limit.
111         *
```

```
112             * @throws IllegalStateException
113             *       if no search-transaction is in progress
114             */
115            void flushToIndexes();
116
117            /**
118             * <b>different from the original Hibernate Search!</b> <br>
119             * <br>
120             * this has to be called when you want to
121             * change the index manually!
122             *
123             * @throws IllegalStateException
124             *       if a search-transaction is already in progress
125             */
126            void beginSearchTransaction();
127
128            /**
129             * <b>different from the original Hibernate Search!</b> <br>
130             * <br>
131             * this has to be called when you want to
132             * change the index manually!
133             *
134             * @throws IllegalStateException
135             *       if no search-transaction is in progress
136             */
137            void rollbackSearchTransaction();
138
139            /**
140             * <b>different from the original Hibernate Search!</b> <br>
141             * <br>
142             * this has to be called when you want to
143             * change the index manually!
144             *
145             * @throws IllegalStateException
146             *       if no search-transaction is in progress
147             */
148            void commitSearchTransaction();
149
150            boolean isSearchTransactionInProgress();
151
152            /**
153             * @throws IllegalStateException
154             *       if search-transaction is still in progress.
155             *       underlying EntityManager is still closed.
156             */
157            void close();
158
159            /**
```

```
160          * Creates a MassIndexer to rebuild the indexes of some
161          * or all indexed entity types. Instances cannot be reused. Any
162          * {@link org.hibernate.search.indexes.interceptor
163          *       .EntityIndexingInterceptor} registered on the entity
164          * types are applied: each instance will trigger an
165          * {@link org.hibernate.search.indexes.interceptor
166          *       .EntityIndexingInterceptor#onAdd(Object)}
167          * event from where you can
168          * customize the indexing operation.
169          *
170          * @param types optionally restrict the operation to
171          *       selected types
172          *
173          * @return a new MassIndexer
174          */
175         MassIndexer createIndexer(Class<?>... types);
176
177 }
```

Listing 43: generic JPA FullTextEntityManager

```
1  /**
2   * Utility class that allows you to access multiple JPA queries at once.
3   * Data is retrieved from the database in batches
4   * and ordered by a given comparator.
5   * No need for messy Unions on the database level! <br>
6   * <br>
7   * This is particularly useful if you scroll all the data
8   * from the database incrementally and if you can
9   * compare in Code.
10  *
11  * @author Martin
12  */
13 public class MultiQueryAccess {
14
15         private final Map<String, Long> currentCountMap;
16         private final Map<String, Query> queryMap;
17         private final Comparator<ObjectIdentifierWrapper> comparator;
18         private final int batchSize;
19
20         private final Map<String, Long> currentPosition;
21         private final Map<String, LinkedList<Object>> values;
22
23         private Object scheduled;
24         private String identifier;
25
26
27         public MultiQueryAccess(
28                 Map<String, Long> countMap,
```

```
29                  Map<String, Query> queryMap,
30                  Comparator<ObjectIdentifierWrapper> comparator,
31                  int batchSize) {
32                  if ( countMap.size() != queryMap.size() ) {
33                          throw new IllegalArgumentException(
34                                  "countMap.size() must be equal " +
35                                          "to queryMap.size()" );
36                  }
37                  this.currentCountMap = countMap;
38                  this.queryMap = queryMap;
39                  this.comparator = comparator;
40                  this.batchSize = batchSize;
41                  this.currentPosition = new HashMap<>();
42                  this.values = new HashMap<>();
43                  for ( String ident : queryMap.keySet() ) {
44                          this.values.put( ident, new LinkedList<>() );
45                          this.currentPosition.put( ident, 0L );
46                  }
47          }
48
49          private static int toInt(Long l) {
50                  return (int) (long) l;
51          }
52
53          /**
54           * increments the value to be returned by {@link #get()}
55           *
56           * @return true if there is a value left to be visited in the database
57           */
58          public boolean next() {
59
60          /*
61           *
62           *
63           * indentation broken to make this readable
64           *
65           *
66           */
67
68  this.scheduled = null;
69  this.identifier = null;
70  List<ObjectIdentifierWrapper> tmp =
71          new ArrayList<>( this.queryMap.size() );
72
73  for ( Map.Entry<String, Query> entry : this.queryMap.entrySet() ) {
74          String identifier = entry.getKey();
75          Query query = entry.getValue();
76          if ( !this.currentCountMap.get( identifier ).equals( 0L ) ) {
```

```
77                  if ( this.values.get( identifier ).size() == 0 ) {
78                          // the last batch is empty. get a new one
79                          Long processed =
80                                  this.currentPosition.get( identifier );
81                          // yay JPA...
82                          query.setFirstResult( toInt( processed ) );
83                          query.setMaxResults( this.batchSize );
84                          @SuppressWarnings("unchecked")
85                          List<Object> list = query.getResultList();
86                          this.values.get( identifier ).addAll( list );
87                  }
88                  Object val = this.values.get( identifier ).getFirst();
89                  tmp.add( new ObjectIdentifierWrapper( val, identifier ) );
90          }
91  }
92  tmp.sort( this.comparator );
93  if ( tmp.size() > 0 ) {
94          ObjectIdentifierWrapper arr = tmp.get( 0 );
95          this.scheduled = arr.object;
96          this.identifier = arr.identifier;
97          this.values.get( this.identifier ).pop();
98          Long currentPosition = this.currentPosition.get( arr.identifier );
99          Long newCurrentPosition =
100                 this.currentPosition
101                         .computeIfPresent( arr.identifier,
102                                 (clazz, old) -> old + 1 );
103         if ( Math.abs( newCurrentPosition - currentPosition ) != 1L ) {
104                 throw new AssertionFailure(
105                         "the new currentPosition count " +
106                         "should be exactly 1 " +
107                         "greater than the old one" );
108         }
109         Long count = this.currentCountMap.get( arr.identifier );
110         Long newCount = this.currentCountMap.computeIfPresent(
111                 arr.identifier, (clazz, old) -> old - 1
112         );
113         if ( Math.abs( count - newCount ) != 1L ) {
114                 throw new AssertionFailure(
115                         "the new old remaining count " +
116                                 should be exactly 1 " +
117                                 "greater than the new one" );
118         }
119 }
120 return this.scheduled != null;
121 }
122
123         /**
124         * @return the current value
```

```
125         */
126         public Object get() {
127                 if ( this.scheduled == null ) {
128                         throw new IllegalStateException(
129                                 "either empty or next() has " +
130                                         "not been called" );
131                 }
132                 return this.scheduled;
133         }
134 
135         /**
136         * @return the identifier of the current value
137         */
138         public String identifier() {
139                 if ( this.identifier == null ) {
140                         throw new IllegalStateException(
141                                 "either empty or next() has " +
142                                         "not been called" );
143                 }
144                 return this.identifier;
145         }
146 
147         public static class ObjectIdentifierWrapper {
148 
149                 public final Object object;
150                 public final String identifier;
151 
152                 public ObjectIdentifierWrapper(Object object,
153                         String identifier) {
154                         this.object = object;
155                         this.identifier = identifier;
156                 }
157 
158         }
159 
160 }
```

Listing 44: MultiQueryAccess.java

Tables

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>
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**)

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur <-Arbeit>

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

$Unterschrift$: $Ort, Datum$: