Logo 1

Logo2

Universität Bayreuth
Fakultät für Informatik

Bachelorarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades
Bachelor / Master of Science

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

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

Version vom: August 2, 2015

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

Zusammenfassung

Abstract

Contents

1 Preface

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

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

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

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

[1] Wikipedia on Java Persistence API, see [1]
[2] Hibernate ORM project homepage, see [9]
[3] Hibernate Search project homepage, see [2]

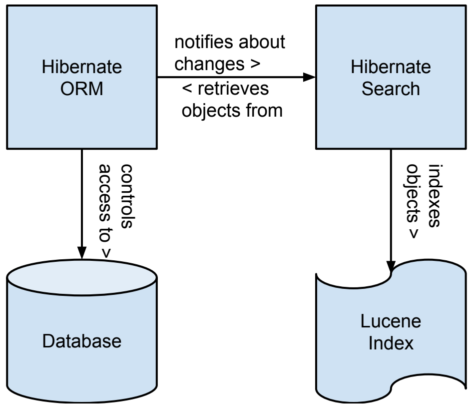

Figure 1: Hibernate Search with Hibernate ORM

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

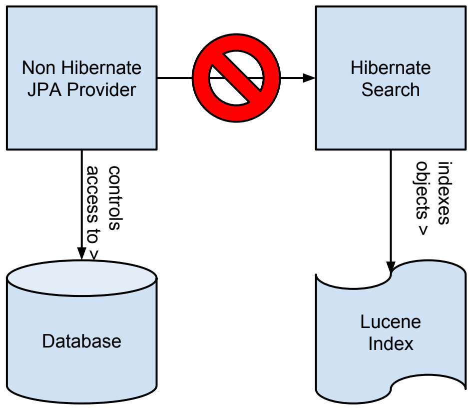

Figure 2: Hibernate Search's incompatibility with other JPA implementations

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

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

[4] official Lucene website, see [18]
[5] ElasticSearch Java API, see [4]
[6] Solr Java API, see [5]

would be a much better suit because of it's design with a entity structure in mind and the automatic index updating feature, if it just were compatible with generic JPA.

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

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

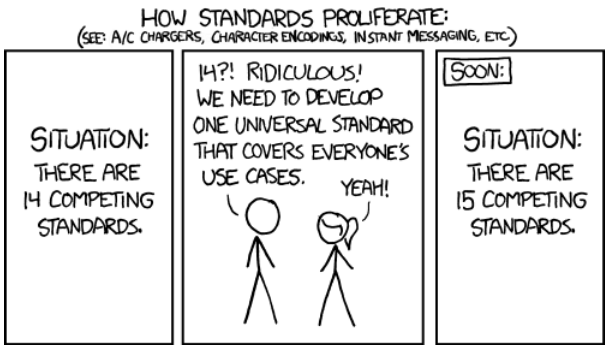

Figure 3: xkcd.com on competing standards [8]

This is why in this thesis we will show how such a generic version can be built. First, we will look at how Hibernate Search's engine can be reused. Then, we will write a standalone version of this engine and finally integrate it with generic JPA.

[7]Hibernate Search GitHub repository, see [13]
[8]xkcd comic #927, see [16]

2 Overview of technologies

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

2.1 Object Relational Mappers

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

```sql
SELECT author.id, author.name, book.id, book.name
FROM author_book, author, author
WHERE author_book.bookid = book.id
AND author_book.authorid = author.id
```

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

This is where Object Relational Mappers (ORM) come into use. They map tables to entity-classes and enable users to write queries against these classes instead of tables. The returned objects are part of a complex object hierarchy and are easier to use from a object oriented point of view.

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

Listing 2: ORM query example

[9]Wikipedia on Object Oriented Programming (OOP), see [6]
[10]Oracle JDBC overview, see [14]
[11]OleDb usage page, see [15]

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 [7]
[13]Hibernate OGM project homepage, see [8]
[14]EclipseLink project homepage, see [12]
[15]EclipseLink project homepage, see [12]
[16]Wikipedia on Java Persistence API, see [1]
[17]Hibernate ORM project homepage, see [9]
[18]EclipseLink project homepage, see [12]
[19]OpenJPA project homepage, see [10]
[20]Wikipedia on Java EE, see [17]

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]:

```sql
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 a good idea.

[21] w3schools on SQL LIKE, see [11]

2.3.1 Lucene

mention current version for each of these?

> Apache Lucene™ is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.[22]

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

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

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

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

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

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

[22] official Lucene website, see [18]
[23] Lucene basic concepts, see [19]

Sowas hier drinlassen? Wenn ja, dann aber auch für Searching
Quelle: http://acupof.blogspot.de/2011/02/lucene-and-hibernate-search-small.html

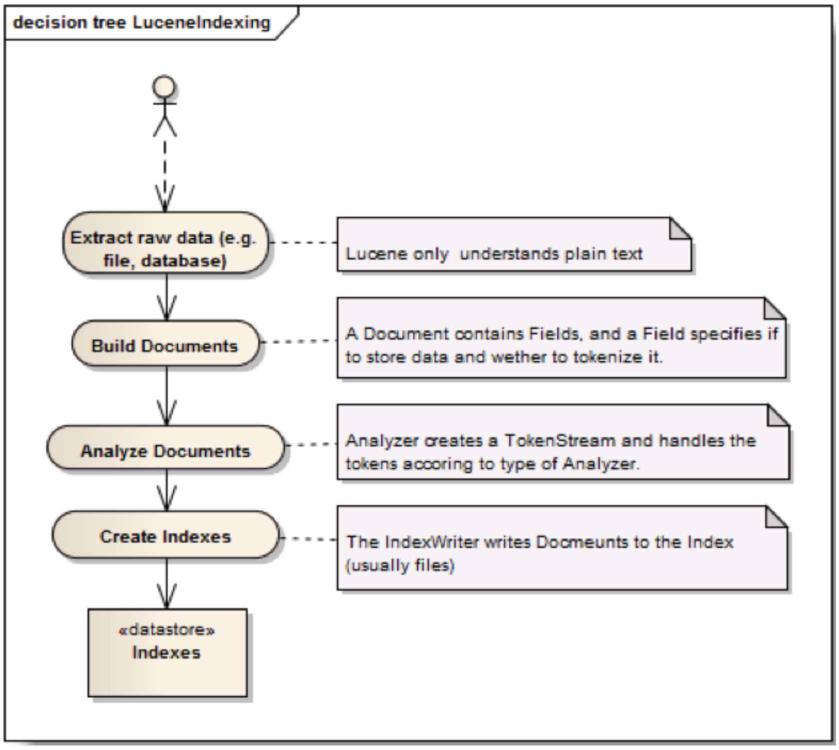

Figure 4: The indexing "pipeline" [24]

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.

[24] Footnote für Bild

2.3.2 Solr

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

2.3.3 ElasticSearch

2.3.4 Hibernate Search

> Hibernate Search transparently indexes your objects and offers fast regular, full-text and geolocation search. Ease of use and easy clustering are core.[25]

some kind of conclusion with a table of features. -> Hibernate Search, aber mit dem Problem von Kompatibilität mit Non Hibernate ORM, mention Compass?

2.3.5 Conclusion: Why a generic Hibernate Search?

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

3 Challenges

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

3.1 The example project

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

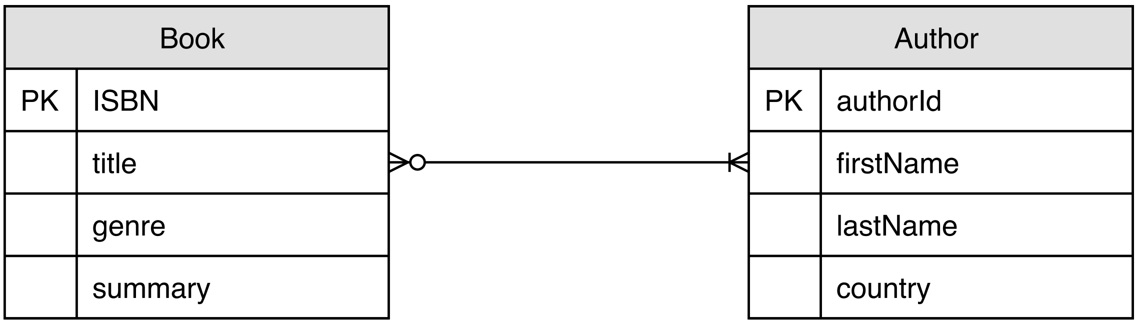

Figure 5: the bookstore entity relationship model

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

```
@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

```
@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 RBDMSs used.

3.2 Indexing & searching

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

3.3 Automatic index updating

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

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

The resulting entities look like this:

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

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

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

```

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

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

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

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

Listing 6: Book.java with Hibernate Search annotations

```
@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

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.

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 map are the following properties:

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[26] (only some Hibernate ORM specific settings cannot be used). Our **StandaloneSearchConfiguration** (appendix listing 20) defaults to "ram" and "4.10.4".

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

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

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

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

[26]Hibernate Search documentation, see [3]

```

//we have to build an integrator here (the builder needs a
//"base integrator" first before we can add index classes)
builder.configuration( searchConfiguration ).buildSearchIntegrator();

indexClasses.forEach( builder::addClass );

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

//use the integrator ...

//close it
searchIntegrator.close();
```

Listing 8: Starting up the engine

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

Indexing objects with **WorkType.INDEX**:

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

Listing 9: Indexing an object with the engine

Updating objects with **WorkType.UPDATE**:

```
Book book = ...;
Transaction tx = new Transaction();
Worker worker = searchIntegrator.getWorker();
```

```
worker.performWork( new Work( book, WorkType.UPDATE ), tx );
tx.commit();
```

Listing 10: Updating an object with the engine

Deleting objects with **WorkType.PURGE**:

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

Listing 11: Deleting an object by id with the engine

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

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.

```
SearchIntegrator searchIntegrator = ...;

HSQuery query = searchIntegrator.createHSQuery();

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

```
                        )
                ).projection(
                        ProjectionConstants.ID
                ).queryEntityInfos();
```

Listing 12: Querying the index with the engine

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

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

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

Listing 13: Extracting info from the results

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 define 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 actual data needed by the calling code 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.

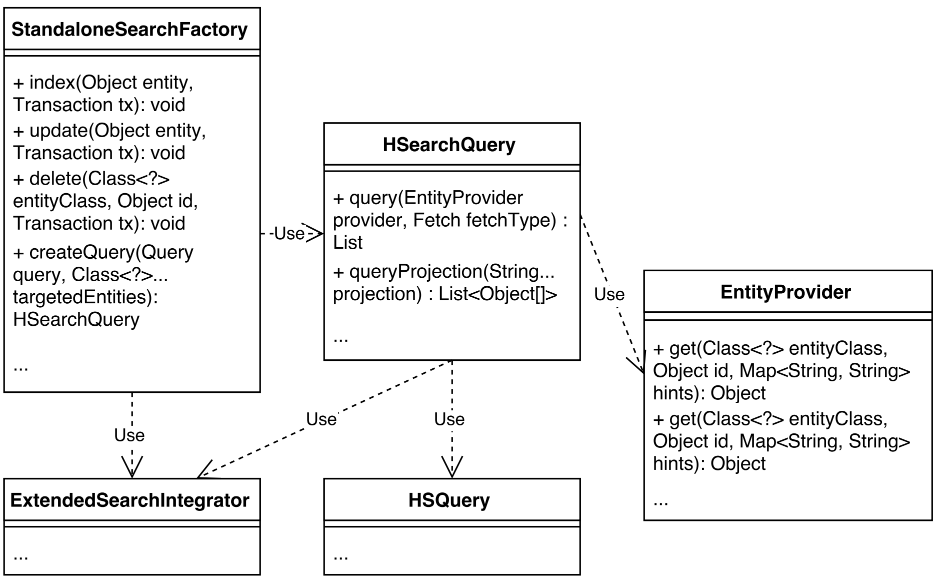

Figure 6: Rough architecture of the standalone (important parts)

4.3.1 Starting the standalone

The startup process of the standalone doesn't differ much from manually using the engine. We still have to use the SearchConfiguration pattern. The only different thing 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.

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

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

StandaloneSearchFactory searchFactory =
                StandaloneSearchFactoryFactory.
                                createSearchFactory(
                                        searchConfiguration,
                                        indexClasses
                                );

//use the searchfactory ...

//close it
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 the Worker pattern from the SearchIntegrator anymore.

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

Listing 15: Indexing an object with the standalone

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

Listing 16: Updating an object with the standalone

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

Listing 17: Deleting an object by id with the standalone

4.3.3 Querying the index

Wie man sieht, ist die query API noch kürzer geworden und wir haben auch die EntityInfos komplett wegrationalisiert

```
StandaloneSearchFactory searchFactory = ...;
EntityProvider entityProvider = new BasicEntityProvider();

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

Listing 18: Querying the index with the standalone

4.4 Standalone integration with JPA interfaces

-> our aim is to be as compatible with the original. -> same interfaces in the same packages, but slight differences

4.4.1 Architecture & differences from the original

zusätzliche annotations, restriktionen!

4.4.2 Implementation Details

-> SubClassSupportInstanceInitializer, MassIndexer, Transaction Management

4.4.3 Usage

4.4.3.1 Starting the JPA integration

4.4.3.2 Indexing, updating and deleting objects from the index

4.4.3.3 Querying the index

4.4.3.4 Index rebuilds

->

4.5 The automatic index updating feature

4.5.1 Overview of possible implementations

4.5.1.1 Synchronous approaches

JPA events

Native integrations with JPA providers

4.5.1.2 Asynchronous approaches

Triggers

4.5.2 Native event integration with JPA providers

4.5.3 Generic approach

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listings

```
public class Transaction implements TransactionContext {

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

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

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

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

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

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

        /**
         * @throws IllegalStateException if already commited/rolledback
         */
        public void rollback() {
                if ( !this.progress ) {
                        throw new IllegalStateException(
                        "can't rollback - " +
                        "No Search Transaction is in Progress!" );
```

```
            }
            this.progress = false;
            this.syncs.forEach( Synchronization::beforeCompletion );

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

}
```

Listing 19: the simple Transaction contract

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

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

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

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

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

```
        }

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

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

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

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

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

```
        }

        @Override
        public Class<?> getClassMapping(String name) {
                return classes.get( name );
        }

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

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

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

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

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

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

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

        @Override
```

```java
	public boolean isTransactionManagerExpected() {
		return this.transactionsExpected;
	}

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

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

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

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

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

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

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

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

}
```

Listing 20: StandaloneSearchConfiguration.java

```
public class BasicEntityProvider implements EntityProvider {

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

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

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

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

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

```java

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

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

}
```

Listing 21: BasicEntityProvider.java

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.

$Unterschrift:$ $Ort, Datum:$