Add full-text to your application with Hibernate Search

Hibernate Search provides an easy way to add full-text search to your application. It uses Apache Lucene or Elasticsearch to provide full-text search capabilities and integrates them with Hibernate ORM. It updates the search indexes transparently and provides a query DSL for full-text queries.

Project setup

Add Hibernate Search to your project

The first thing you need to do, if you want to add Hibernate Search to your project is to add the required libraries to your project. These are the hibernate-search-orm.jar and if you want to use it with JPA also the hibernate-entitymanager.jar.

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-orm</artifactId>
   <version>5.6.0.Final</version>
</dependency>
```

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>${hibernate.version}</version>
</dependency>
```

Configuration

You don’t need to provide any configuration when you start to use Hibernate Search. The default values provide a good starting point for most standard applications.

But I recommend to use the filesystem DirectoryProvider in the beginning. It stores the Lucene indexes in the file system which allows you to easily inspect them and get a better understanding of your system. When you’re familiar with Hibernate Search and Lucene, you should also have a look at the other supported DirectoryProviders.

```
<persistence>
   <persistence-unit name="my-persistence-unit">
        ...

        <properties>
          ...


                    <property name =
                              "hibernate.search.default.directory_provider"
                              value="filesystem"/>
                    <property name =
                              "hibernate.search.default.indexBase"
                              value="./lucene/indexes"/>
        </properties>
    </persistence-unit>
</persistence>
```

Index entity attributes

Indexing one of your entities requires 2 things:

1. You need to annotate the entitiy with *@Indexed* to tell Hibernate Search to index the entity.
2. You need to annotate the fields you want to index with the *@Field* annotation. This annotation also allows you to define how the attributes will be indexed. I will get into more detail about that in one of the following blog posts.

```
@Indexed
@Entity
public class Tweet {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "id", updatable = false, nullable = false)
        private Long id;

        @Column
        @Field
        private String userName;

        @Column
        @Field
        private String message;

        …
}
```

Perform a simple full-text search

Similar to a search on Google, you can now use Hibernate Search to do a full-text search on the messages of these tweets. The following code snippet shows a query that searches for the words “validate” and “Hibernate” in the messages of the tweets.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

FullTextEntityManager fullTextEm = Search.getFullTextEntityManager(em);
QueryBuilder tweetQb = fullTextEm.getSearchFactory()
        .buildQueryBuilder()
        .forEntity(Tweet.class)
        .get();
Query fullTextQuery = tweetQb.keyword()
        .onField(Tweet_.message.getName())
        .matching("validate Hibernate")
        .createQuery();
List<Tweet> results = fullTextEm
        .createFullTextQuery(fullTextQuery, Tweet.class)
        .getResultList();
```

In the first step, you need to get a *FullTextEntityManager*. It extends the *EntityManager* interface with full-text search capabilities and allows you to create a *QueryBuilder* for the entity class you’re searching. In this example, I create a *QueryBuilder* for my *Tweet* entity. You then use the *QueryBuilder* to define your query. I want to do a keyword search on the message field. That searches the index with message attributes for one or more words. In this case, I’m searching for the words “validate” and “Hibernate”. Then I create a query and provide it to the *createFullTextQuery* mehod.

This method returns a *FullTextQuery* interface which extends JPA's *Query* interface. And then I call the *getResultList* method to execute the query and get a List of results.

Internally, this query gets executed in 2 steps: First Hibernate Search uses the Lucene index to perform the full-text search and return the primary keys of the matching entities. Then Hibernate ORM uses the primary keys to selects the entities.

Similar to a Google search, this queries also returns documents that contain only one of the search terms. But as you can see in the log output, the Tweet with the message "How to automatically validate entities with Hibernate Validator BeanValidation" received the better ranking because it contained both search terms.

```
15:04:29,704 DEBUG SQL:92 - select this_.id as id1_0_0_, this_.message as message2_0_0_, this_.postedAt as postedAt3_0_0_, this_.url as url4_0_0_, this_.userName as userName5_0_0_, this_.version as version6_0_0_ from Tweet this_ where (this_.id in (?, ?))

15:04:29,707  INFO TestSearchTweets:55 - Tweet [id=3, postedAt=2017-02-02 00:00:00.0, userName=thjanssen123, message=How to automatically validate entities with Hibernate Validator BeanValidation, url=http://www.thoughts-on-java.org/automatically-validate-entities-with-hibernate-validator/, version=0]

15:04:29,707  INFO TestSearchTweets:55 - Tweet [id=2, postedAt=2017-01-24 00:00:00.0, userName=thjanssen123, message=5 tips to write efficient queries with JPA and Hibernate, url=www.thoughts-on-java.org/5-tips-write-efficient-queries-jpa-hibernate/, version=0]
```