GETTING STARTED WITH

Hibernate Search

UPDATED BY SANNE GRINOVERO

SEARCH YOUR DATABASE!

Hibernate Search is an extension to Hibernate ORM that adds powerful full-text query capabilities, with an API consistent with Hibernate and JPA. It is typically used to implement "full-text search," such as matching free text input provided by humans.

Hibernate Search can be used to create a search experience similar to what one would expect from Google or Amazon, and it can also be used to run queries that combine natural language phrases and spatial filtering. Additionally, it can be used simply to boost the performance of some queries: the dedicated index generated by Hibernate Search can be a more efficient way to implement any text-based query. As you get more familiar with Hibernate Search, you can even replace some relational queries with more efficient alternatives. You'll only have to define the structure of your index declaratively, as index writes and updates are handled automatically.

Hibernate Search depends on Apache Lucene, the leading full-text search engine library hosted at the Apache Software Foundation. This Refcard explains how to Map entities, set up basic configuration, build indexes, query them and examine their contents.

GET STARTED

In order to use Hibernate Search, you should be familiar with the Hibernate ORM APIs, or the JPA APIs, as these will be used to update your databases and the index. You won't need to learn a new API to update the index as this will be kept in sync with your database automatically. It does however provide new querying methods that expose the full-text capabilities—these are not covered by the traditional Hibernate nor the JPA APIs.

Similarly, you should be familiar with the JPA mapping annotations, which are equally extended with additional full-text mapping annotations to define which fields you want to be indexed, and with which indexing options.

Download Hibernate Search from hibernate.org/search/downloads/ or use the Maven coordinates:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-orm</artifactId>
   <version>5.5.5.Final</version>
</dependency>
```

Hibernate Search is available in Maven Central so you typically don't need to enable any specific Maven repository.

The version of Hibernate Search which you will be using does not necessarily match the version of Hibernate ORM which you might already be using, so you need to pick a compatible version.

For this guide we'll use the following combination of versions which are compatible—as you can typically find out yourself by reading either the Maven pom of a release or its README file:

- Hibernate Search 5.5.5.*Final*
- Hibernate ORM (and EntityManager) 5.0.11.*Final*
- Apache Lucene 5.3.1
- Hibernate Common Annotations 5.0.1.*Final*

Hibernate Search is not released as part of Hibernate ORM, so its version number will often be different—do not expect them to match. Read the documentation to pick a compatible version, or let your dependency management tool pick the right versions for you.

Hibernate Search does not require a container and can run in Java SE. It can also run in Java EE containers, OSGi containers, Apache Tomcat, within Spring or any other environment able to run Hibernate ORM; WildFly users won't need a separate download as it is included in the popular Application Server.

CONFIGURATION PROPERTIES

No configuration properties are *required* to get started: Hibernate Search will automatically integrate with your Hibernate ORM Session(s) or EntityManager(s) when it is found on the classpath.

The following table lists some of the most useful configuration properties. None of them are required:

`hibernate.search.default.indexBase`	When using "filesystem" as directory_provider: defines the filesystem path in which to store the index.
`hibernate.search.indexing_strategy`	Set it to "manual" to disable automatic indexing.
`hibernate.search.default.indexmanager`	Set it to "near-real-time" to use Lucene's NRT capabilities. Very efficient but might lose some writes in case the JVM terminates; hence disabled by default.
`hibernate.search.default.worker.execution`	"sync" is the default. Set to "async" to perform the index write operations in background.
`hibernate.search.default.index_flush_interval`	The interval (in ms) between flushes of write operations to the index. Only applies if worker.execution is configured as async.
`hibernate.search.error_handler`	Set to the fully qualified classname of a custom "ErrorHandler" implementation to handle all indexing errors.
`hibernate.search.default.indexwriter.ram_buffer_size`	Set it to 64 or more, if you can afford dedicating that memory (in MBs) to write buffers.

The reference documentation will list many more properties, but for most use cases these are all you need to know.

These configuration properties can be defined in any place were you normally define Hibernate configuration properties, including System Properties, or the *hibernate.properties* resource file.

For example, they could be listed in your *META-INF/persistence.xml* , just like other Hibernate properties:

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="demo1">
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.H2Dialect" />
      <property name="hibernate.connection.provider_class"
        value="org.hibernate.hikaricp.internal.
          HikariCPConnectionProvider" />
      <property name="hibernate.hikari.dataSourceClassName"
                value="org.h2.jdbcx.JdbcDataSource" />
      <property name="hibernate.hikari.dataSource.url"
                value="org.h2.jdbcx.JdbcDataSource" />
      <property name="hibernate.hikari.dataSource.user"
                value="sa" />
      <property name="hibernate.hikari.dataSource.password"
                value="" />
      <property name="hibernate.search.default.directory_
                  provider"
                value="filesystem" />
      <property name="hibernate.search.default.indexBase"
                value="/home/me/tests/index1" />
      <property name="hibernate.search.default.indexmanager"
                value="near-real-time" />
    </properties>
  </persistence-unit>
</persistence>
```

In a typical Hibernate Search powered application you'll have multiple indexes, as it is often a good idea to index different entities separately.

Each index is identified by having a name, and some of the properties can be applied to a specific index; you can easily recognize the configuration properties which can be customized on a per-index case as they have the "hibernate.search.default" prefix; the *default* means it applies to all indexes, unless overridden. You can replace the "default" keyword with a specific index name to tune it independently.

For example, you can customize the storage of a specific index:

```
hibernate.search.default.directory_provider = filesystem
hibernate.search.default.indexBase = /home/me/tests/allindexes
hibernate.search.ANIMALS.indexBase = /fastdrive/AnimalsIndex
```

INDEXING GUIDELINES

A Lucene index is an extremely powerful tool, but to make the most of it you have to appreciate its differences with relational databases—it's a new world. Some classify Lucene as a form of NoSQL.

The most obvious drawback is that you'll have to manage and maintain an index, typically stored on fast file systems, in addition to managing your relational databases; yet, many applications have found the benefits outweighed the effort to achieve them.

You'll never interact with the Lucene index writing process, you only have to declare how each entity needs to be transformed: when you make a change to your entities and the transaction is successfully committed to the database, Hibernate Search will transparently update the Lucene index to keep the two in sync.

Some hints to get you in the right mindset:

- the way you index your data will determine how you can query it.
- an *indexed* property can be searched, but not read back in its original form unless it's also *stored*.
- only index the bare minimum you need for your queries: a small index is a fast index.
- the index is not relational: you can't run "join" operators in the query so make sure to include—flattened—all information you need to filter.

The typical Hibernate Search query is run on the index, which is very efficient, and when the matches are identified the actual entities are loaded from the database within your usual transaction scope.

PROJECTION QUERY

Sometimes you might want to *store* some properties in the index too; this will allow you to run a Projection Query, which returns properties directly from the index rather than from the database. This might be efficient to load a list of previews—such as item names—while skipping the database reads, but while it might be faster you trade-off on consistency as no transactions are involved so you can't load managed entities via a Projection Query.

MAPPING ENTITIES

Hibernate Search expands the well known Hibernate and JPA annotations to let you define how your Java POJOs should be indexed.

An entity will be indexed if it's marked with the `@Indexed` annotation; the second step will be to pick properties for index inclusion via the `@Field` annotation; an entry in the Lucene index is represented by a Lucene Document, which has several named fields. By default an entity property will be mapped to a Lucene Field with the same name.

BASIC MAPPING EXAMPLE

```
import java.time.LocalDate;
import javax.persistence.Entity;
import javax.persistence.Id;
import org.hibernate.search.annotations.Field;
import org.hibernate.search.annotations.Indexed;

@Entity
@Indexed
public class Explorer {
    @Id int id;
    @Field String name;
    @Field LocalDate dateOfBirth;
}
```

The `@Indexed` annotation allows—among others—to pick an index *name*.

The `@Field` annotation has several interesting attributes:

- *name*: to customize the name of the Field in the Lucene Document.
- *store*: allows to store the value in the index, allowing loading over Projection
- *analyze*: when disabled, the value will be treated as a single keyword rather than being processed by the Analyzer chain (see Analyzers).
- *bridge*: allows customizing the encoding/decoding to and from the index format. Required only for custom types.

MAKE IT SORTABLE

If you want to be able to sort on a field, you have to prepare the index format for the operation using the `@Sortable` field:

```
@Entity
@Indexed
public class Explorer {
    @Id int id;
    @SortableField
    @Field
    String name;
    @Field LocalDate dateOfBirth;
}
```

INDEXING SPATIAL LOCATIONS

Using the `@Spatial` annotation, you can index pairs of coordinates to search for entries within a distance from a point, or sort by distance from a point.

This feature should not be confused with Hibernate Spatial, which integrates with RDBMS-based types; the Hibernate Search support for Spatial integration is limited to distances from points but shines when this factor has to be combined with a full text query.

For example you might want to search for a restaurant by approximate name *and* by not being too far away:

```
@Indexed @Spatial @Entity
public class Restaurant {
    @Id id;
    @Field name;
    @Latitude
    Double latitude;

    @Longitude
    Double longitude;
}
```

The Spatial feature of Hibernate Search is extensive and we suggest reading the reference documentation (Chapter 9) to better understand its features. For example, it can use different indexing techniques, you can have multiple sets of spatial coordinates per entity, or represent points by implementing the *org.hibernate.search.spatial.Coordinates* interface.

INDEX ASSOCIATIONS

The index technology is not a good fit for "join queries," and generally to represent associations. Hibernate Search allows to embed fields from related entities into its main document, but remember that while these fields are useful to narrow down your queries it is not a true association: you're in practice creating a denormalized representation of your object graph.

In the following example, the *Movie* entity will generate an index having two indexed fields: *title* and *actors.name*, in addition to the id field which is always indexed. The *actors.name* field might be repeated multiple times in the index, and therefore match simultaneously various different names.

```
@Indexed @Entity
public class Movie {
    @Id @GeneratedValue private Long id;
    @Field private String title;
    @ManyToMany
    @IndexedEmbedded
    private Set<Actor> actors;
    ...
}

@Entity
public class Actor {
    @Id @GeneratedValue private Integer id;
    @Field private String name;
    @ManyToMany(mappedBy="actors")
    @ContainedIn
    private Set<Movie> movies;
    ...
}
```

QUICK REFERENCE OF ALL HIBERNATE SEARCH ANNOTATIONS

Table of all Hibernate Search annotations—the package name is *org.hibernate.search.annotations*:

`@AnalyzerDefs`	Contains multiple `@AnalyzerDef` annotations.
`@AnalyzerDiscriminator`	Used to choose an Analyzer dynamically based on an instance's state.
`@Analyzer`	Defines which Analyzer shall be used; can be applied on the entity, a method or attribute, or within a `@Field` annotation.

`@Boost`	Statically "boost" a property or an entity, to have more (or less) influence on the score during queries.
`@CalendarBridge`	Set indexing options on a property of type Calendar.
`@CharFilterDef`	Create a character filter definition; used as parameter of `@AnalyzerDef`. You can list multiple character filters, they will be applied in order.
`@ClassBridge`	Customizes the conversion process from entity to indexed fields by implementing a "`org.hibernate.search.bridge.FieldBridge`".
`@ClassBridges`	Contains multiple `@ClassBridge` annotations.
`@ContainedIn`	Marks the inverse association of an `@IndexedEmbedded` relation.
`@DateBridge`	Set indexing options on a property of type Date.
`@DocumentId`	Override the property being used as primary identifier in the index. Defaults to the property with JPA's `@Id`.
`@DynamicBoost`	Similar to `@Boost` but allows dynamic selection of a boost value, based on the instance properties.
`@Facet`	Marks a property to allow faceting queries.
`@Facets`	Contains multiple `@Facet` annotations.
`@FieldBridge`	Specifies a custom `FieldBridge` implementation class to use to convert a property type into its index representation.
`@Field`	Marks an entity property to be indexed. Supports various options to customize the indexing format.
`@Fields`	Contains multiple `@Field` annotations. Indexes the annotated property multiple times: each `@Field` shall have a different name and can have different options.
`@FullTextFilterDef`	Define a filter, which can be applied to full-text queries.
`@FullTextFilterDefs`	Define multiple filters by repeating `@FullTextFilterDef`
`@IndexedEmbedded`	Recurses into associations to include indexed fields in the parent index document. Allows a prefix name, and strategies to limit the recursion. Often to be used with `@ContainedIn` on the inverse side of the relation.
`@Indexed`	Marks which entities shall be indexed; allows to override the index name. Only `@Indexed` entities can be searched.
`@Latitude`	Marks a numeric property as being the "latitude" component of a coordinates pair for a spatial index.
`@Longitude`	Marks a numeric property as being the "longitude" component of a coordinates pair for a spatial index.
`@NumericField`	Applies on a property having a numeric value, allows customizing the index format such as the precision of its representation.
`@NumericFields`	Allows indexing a single numeric property into multiple different numeric index fields, and customize each, by listing several `@NumericField` annotations.
`@Parameter`	Used to pass an initialization parameter to tokenizers and filters in the scope of an `@AnalyzerDef`, or to pass parameters to your custom bridge implementations in the scope of a `@FieldBridge`.
`@SortableField`	Marks a property so that it will be indexed in such way to allow efficient sorting on it.
`@SortableFields`	When a property is indexed generating multiple fields, this allows to choose multiple of these fields to be good candidates for sorting. Container for `@SortableField`.
`@Spatial`	Defines a named couple of coordinates. You can have it on an entity, or on a getter returning an implementation of **org.hibernate.search.spatial.Coordinates**
`@Spatials`	Used to define multiple `@Spatial` pairs of coordinates; assign a unique name to each of them.
`@TikaBridge`	Applied on an indexed property it will be treated as a resource processed with Apache Tika to extract meaningful text from it, for example metadata from music or plain text from office documents. The property can point to the resource (*String* containing a path, *URI*) or contain the resource (*byte[]*, *java.sql.Blob*).
`@TokenFilterDef`	Defines the Token Filter(s) in an `@AnalyzerDef`. You can list multiple filters, they will be applied in order.
`@TokenizerDef`	Chooses the Tokenizer to be used in an `@AnalyzerDev`.

ANALYZERS

The Analyzer is the Lucene component that processes a stream of text into signals of interest to be stored in the index.

A simple Analyzer, for example, could apply this process:

1. split the input string on any whitespace character
2. lowercase each of these tokens
3. replace accented characters with the simpler form

The problem might seem simple when limited to whitespace chunking and lowercasing, but your application can provide much better results—and therefore be much more useful—if you adjust your analyzer chain to your specific needs.

For example you might want to use N-Grams to minimize the effect of user typos, an exotic language might require special handling for some lower casing, you might want to highlight specific keywords, cleanup HTML tags, replace each generic noun with a canonical synonym, expand domain specific acronyms, apply stemming.

While an internet search is targeting a wide audience and indexing disparate content, when building your own application you can embed it with a domain specific, specialized search engine just by picking the right analyzers.

The Apache Lucene project documentation provides a great introduction to the Analyzer chain here: lucene.apache.org/core/5_3_0/core/org/apache/lucene/analysis/package-summary.html

Do not be scared by the examples of how to use an Analyzer though, as with Hibernate Search you merely have to pick the Tokenizer(s) and Filter(s) implementations you want to use.

Let's now make a fairly complex example. We will define two analyzers, named "en" and "de". We define these using the `@AnalyzerDef` annotation on an indexed entity, and this makes both Analyzers available to Hibernate Search, and we can refer to each by name:

```
@Entity
@Indexed
@AnalyzerDefs({
    @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory =
        StandardTokenizerFactory.class),
    filters = {
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory =
        SnowballPorterFilterFactory.class, params = {
        @Parameter(name = "language", value = "English")
    })
}),

@AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory =
    StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory =
            LowerCaseFilterFactory.class),
        @TokenFilterDef(factory =
            GermanStemFilterFactory.class)
    })
})

public class Article {
    private Integer id;
    private String language;
    private String text;
    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Field(store = Store.YES)
    @AnalyzerDiscriminator(impl =
        LanguageDiscriminator.class)
    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }

    @Field(store = Store.YES)
        public String getText() {
        return text;
    }
    ...
}
```

The “en” Analyzer is defined as: tokenize all input using the *StandardTokenizerFactory*, then process it with *LowerCaseFilterFactory* and finally process it with the *SnowballPorterFilterFactory*, initialized with the “language” parameter set to “English”.

The “de” Analyzer is similar as it also begins with a *StandardTokenizerFactory* followed by lower casing thanks to *LowerCaseFilterFactory*, but is then processed by the *GermanStemFilterFactory*, which requires no parameters.

In the above example, we also introduced the rather advanced annotation `@AnalyzerDiscriminator`. This annotation refers to a custom implementation of *org.hibernate.search.analyzer.Discriminator*; a suitable implementation for the above example would have to return either “de” or “en” and this would allow the *Article* entity to be processed with the corresponding Analyzer depending on its language property:

```
public class LanguageDiscriminator implements Discriminator {
    @Override
    public String getAnalyzerDefinitionName(Object value,
        Object entity, String field) {
        if ( value == null ||
            !( entity instanceof Article ) ) {
            return null;
        }
        return (String) value;
    }
}
```

EXECUTING QUERIES

While indexing operations happen transparently and you only had to use annotations to declare what and how you want things indexed, to run queries you need to invoke methods on the *org.hibernate.search.FullTextSession* (extending the Hibernate native *org.hibernate.Session*) or *org.hibernate.search.jpa.FullTextEntityManager* (extending the JPA javax.persistence.EntityManager). You can use either of them as you prefer.

When running a full-text query you need to take into consideration the indexing options, and sometimes also consider the Analyzer being used on each field.

It all starts from a `FullTextSession`:

```
FullTextSession fullTextSession =
    Search.getFullTextSession(session);
```

You can now obtain a `QueryBuilder`:

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity(Movie.class)
    .get();
```

Now create a Lucene Query by specifying:

1. the type of query
2. which index fields it targets
3. what you’re looking for

```
org.apache.lucene.search.Query query = builder.keyword()
    .onField("title")
    .matching(queryString)
    .createQuery();
```

Transform the Lucene Query into an Hibernate Query:

```
org.hibernate.search.FullTextQuery ftq =
    fullTextSession.createFullTextQuery(query, Movie.class);
```

Execute the query:

```
List<Movie> results = hibQuery.list();
```

This will return a list of all Movie instances matching the query, however the `keyword()` query type is not very flexible: it will only match movie titles which have a token which matches exactly the `queryString` parameter. This is useful mostly to search for exact terms (keywords), which are often indexed but not analyzed.

The full-text engine is much more impressive when using the other Query types it can generate, and combine them:

`builder.bool()`	Starts the context to generate boolean queries.
`builder.facet()`	Create a faceting query (is run differently as it returns a more complex type)
`builder.keyword()`	Create a Lucene TermQuery: matches exactly a term. Analysis is disabled.
`builder.moreLikeThis()`	Starting context to create a "More Like This" query, which takes an entity instance as an example of what to look for.
`builder.phrase()`	To create a `org.apache.lucene.search.PhraseQuery`, a powerful full-text matching query for which relative term positions matter.
`builder.range()`	Creates `org.apache.lucene.search.TermRangeQuery` queries
`builder.spatial()`	To search by disance on coordinates defined by `@Spatial`
`-no builder-`	You don't have to use the builder: you can create and use any org.apache.lucene.search.Query implementation.

To sort results just use the Lucene API, but set the sorting strategy on the Hibernate FullTextQuery:

```
ftq.setSort(new Sort(new SortField("price",
    SortField.Type.INT)));
```

Make sure to pick a compatible sorting type to match the field, and to annotate the field with `@SortableField`.

Many more advanced query capabilities are documented in the reference documentation. Among these, you might want to check:

- Projection queries: load values directly from the index
- Spatial queries: sort by distance
- Combine queries with boolean operators
- Project the EXPLANATION of the result scores and Document Ids for debugging purposes
- Project the distance from the Spatial point
- Faceting Queries
- Enabling filters
- Customize the scoring

SYNCHRONIZING THE INDEX WITH THE DATABASE

Hibernate Search aims to keep the index content aligned with the database content automatically, but in some cases you will need to explicitly request an index rebuild.

Re-indexing is typically needed in the following cases:

- You are making changes to the Hibernate Search annotations and re-deploy a new version of your application: the existing index will reflect the outdated "schema".
- The database was updated using a different tool, for example a backup was restored: the index will refer to the previous data.
- You disabled the Hibernate Search listeners explicitly as you prefer to rebuild the index periodically (some people do this every night).
- Some disaster happened and the index was lost.

Rebuilding the index is simple:

```
fullTextSession.createIndexer().startAndWait();
```

Remember this operation can take a while as it will have to iterate on all your indexed entities from the database; this is a multi-threaded operation designed to use multiple database connections and CPUs, so make sure that your servers and databases are ready for the additional load.

Spend some time experimenting with the various tuning options: this process can take days, or just seconds! Reading the chapters about performance tuning of the reference documentation might be worth your time.

TOOLING

On top of the "traditional" Hibernate Tools available for Eclipse, IntelliJ IDEA and NetBeans, when using Hibernate Search it is handy to be able to inspect the Apache Lucene index, so that you can verify the index structure, look at how your data is getting encoded and experiment with queries directly on the index.

LUKE

Luke is a great standalone Java graphical application which lets you analyze a Lucene index; originally created by Andrzej Bialecki, there are now multiple forks, but the most popular—and probably the better maintained—is the GitHub fork from Dmitry Kan at github.com/DmitryKey/luke.

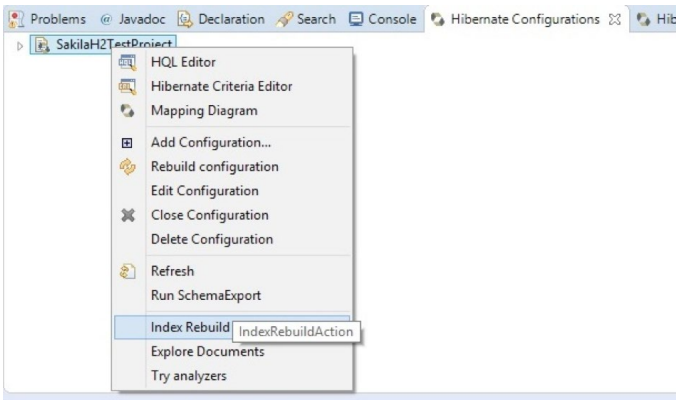
HIBERNATE SEARCH PLUGIN FOR ECLIPSE

Dmitry Bocharov was inspired by Luke and created an Eclipse plugin version which integrates with the Hibernate Tools for Eclipse.

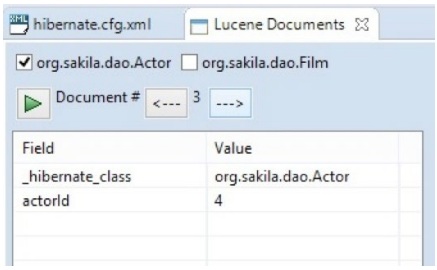
This is actively evolving from its initial status of Google Summer of Code work; while Luke offers more features and is the most mature, the new Eclipse plugin aims specifically at Hibernate Search and a nice integration within Eclipse.

You can find it in the Eclipse Marketplace; remember to install the Hibernate Tools first as it is a dependency, or use JBoss Developer Studio which comes preloaded with it.

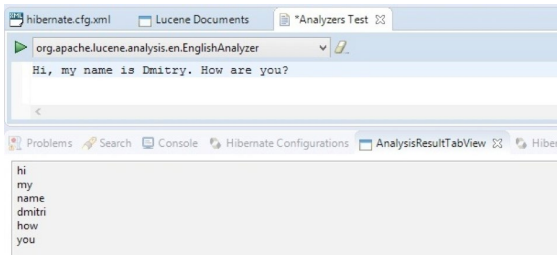
It provides a handy menu option to rebuild the index:



A new view to explore the Lucene index:



And an area to experiment with your custom defined Analyzers:



COMMUNITY RESOURCES

This table shows links to documentation and the links to access the friendly OSS community.

TOPIC	URL
Hibernate Search homepage	hibernate.org/search
GitHub Main repository	github.com/hibernate/hibernate-search
Apache Lucene homepage	lucene.apache.org/core
Hibernate team blog	in.relation.to
Mailing lists, IRC channels, forums	hibernate.org/community
JIRA	hibernate.atlassian.net/projects/HSEARCH
Tag on Stack Overflow	hibernate-search

ABOUT THE AUTHOR

SANNE GRINOVERO has been a member of the Hibernate team for almost ten years, always having an interest in performance, scalability, and integrations with Apache Lucene.

After falling in love with Hibernate Search as a user first, then as contributor, Sanne now leads this open source project in his role of Principal Software Engineer at Red Hat.

Curiosity for distributed systems led him to contribute to the Infinispan project; he co-founded the Hibernate OGM project to make NoSQL technologies more approachable to the JavaEE ecosystem and to the WildFly application server.

He has lived in the Netherlands, Italy, Santo Domingo, Chile, Portugal and currently hacks in London.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

BROUGHT TO YOU IN PARTNERSHIP WITH
redhat.