

# Reference Documentation

1.0.0

Copyright (c) 2004-2005 Shay Banon (kimchy), Alan Hardy

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

---

# Table of Contents

Preface .....	
I. Compass::Core .....	
<b>1. Introduction</b> .....	
1.1. Overview .....	2
1.2. Template and Callback .....	2
<b>2. Configuration</b> .....	
2.1. Introduction .....	4
2.2. Programmatic Configuration .....	4
2.3. XML Configuration .....	5
2.3.1. Schema Based Configuration .....	5
2.3.2. DTD Based Configuration .....	7
2.4. Obtaining a Compass reference .....	8
<b>3. Search Engine</b> .....	
3.1. Introduction .....	9
3.2. Alias, Resource and Property .....	9
3.3. Creating Resource and Property .....	9
3.3.1. Boosting Resource and Property .....	11
3.4. Analyzers .....	11
3.5. Index Structure .....	12
3.6. Transaction .....	12
3.6.1. Locking .....	12
3.6.2. read_committed .....	12
3.6.3. serializable .....	13
3.6.4. batch_insert .....	13
3.7. Optimizers .....	13
3.7.1. Scheduled Optimizers .....	13
3.7.2. Aggressive Optimizer .....	13
3.7.3. Adaptive Optimizer .....	13
3.7.4. Null Optimizer .....	14
<b>4. OSEM - XML</b> .....	
4.1. Introduction .....	15
4.2. Searchable Classes .....	15
4.2.1. Implement a Default Constructor .....	16
4.2.2. Provide Property Identifier(s) .....	16
4.2.3. Declare Accessors and Mutators (Optional) .....	16
4.2.4. Implementing equals() and hashCode() .....	16
4.3. Mapping .....	16
4.3.1. compass-core-mapping .....	17
4.3.2. class .....	18
4.3.3. contract .....	19
4.3.4. id .....	20
4.3.5. property .....	21
4.3.6. analyzer .....	22
4.3.7. meta-data .....	23
4.3.8. component .....	24
4.3.9. reference .....	25
4.3.10. parent .....	26
4.3.11. constant .....	26

<b>5. OSEM - Annotations</b> .....	
5.1. Introduction .....	28
5.2. Searchable Classes .....	28
5.2.1. Implement a Default Constructor .....	29
5.2.2. Provide Property Identifier(s) .....	29
5.2.3. Declare Accessors and Mutators (Optional) .....	29
5.2.4. Implementing equals() and hashCode() .....	29
5.3. Mapping Annotations .....	29
5.3.1. @Searchable .....	29
5.3.2. @SearchableId .....	29
5.3.3. @SearchableProperty .....	30
5.3.4. @SearchableComponent .....	30
5.3.5. @SearchableReference .....	30
5.4. Configuration Annotations .....	31
<b>6. XSEM - Xml to Search Engine Mapping</b> .....	
6.1. Introduction .....	32
6.2. Xml Object .....	32
6.3. Xml Content Handling .....	32
6.4. Raw Xml Object .....	34
6.5. Mapping Definition .....	34
6.5.1. xml-object .....	35
6.5.2. xml-id .....	36
6.5.3. xml-property .....	37
6.5.4. xml-analyzer .....	38
6.5.5. xml-content .....	39
<b>7. Resource Mapping</b> .....	
7.1. Introduction .....	40
7.2. Mapping Declaration .....	40
7.2.1. resource .....	41
7.2.2. resource-contract .....	42
7.2.3. resource-id .....	42
7.2.4. resource-property .....	43
7.2.5. resource-analyzer .....	44
<b>8. Common Meta Data</b> .....	
8.1. Introduction .....	45
8.2. Common Meta Data Definition .....	45
8.3. Using the Definition .....	46
8.4. Common Meta Data Ant Task .....	47
<b>9. Transaction</b> .....	
9.1. Introduction .....	48
9.2. Session Lifecycle .....	48
9.3. Local Transaction .....	48
9.4. JTA Synchronization Transaction .....	48
<b>10. Working with objects</b> .....	
10.1. Introduction .....	50
10.2. Making Object/Resource Searchable .....	50
10.3. Loading an Object/Resource .....	50
10.4. Deleting an Object/Resource .....	50
10.5. Searching .....	50
10.5.1. Query String Syntax .....	51
10.5.2. CompassHits, CompassDetachedHits & CompassHitsOperations .....	51
10.5.3. CompassQuery and CompassQueryBuilder .....	52

10.5.4. CompassHighlighter .....	53
II. Compass::Vocabulary .....	
<b>11. Introduction</b> .....	
<b>12. Dublin Core</b> .....	
III. Compass::Gps .....	
<b>13. Introduction</b> .....	
13.1. Overview .....	58
13.2. CompassGps .....	58
13.2.1. SingleCompassGps .....	58
13.2.2. DualCompassGps .....	59
13.3. CompassGpsDevice .....	59
13.3.1. MirrorDataChangesGpsDevice .....	59
13.4. Programmatic Configuration .....	60
13.5. Building a Gps Device .....	60
<b>14. JDBC</b> .....	
14.1. Introduction .....	61
14.2. Mapping .....	61
14.2.1. ResultSet Mapping .....	61
14.2.2. Table Mapping .....	62
14.3. Mapping - MirrorDataChanges .....	63
14.3.1. ResultSet Mapping .....	63
14.3.2. Table Mapping .....	63
14.3.3. Jdbc Snapshot .....	63
14.4. Resource Mapping .....	64
14.5. Putting it All Together .....	64
<b>15. Hibernate</b> .....	
15.1. Introduction .....	65
15.2. Configuration .....	65
15.3. Index Operation .....	65
15.4. Real Time Data Mirroring .....	65
<b>16. JPA (Java Persistence API)</b> .....	
16.1. Introduction .....	67
16.2. Configuration .....	67
16.3. Index Operation .....	67
16.4. Real Time Data Mirroring .....	68
<b>17. JDO (Java Data Objects)</b> .....	
17.1. Introduction .....	69
17.2. Configuration .....	69
17.3. Index Operation .....	69
17.4. Real Time Data Mirroring .....	69
<b>18. OJB (Object Relational Broker)</b> .....	
18.1. Introduction .....	70
18.2. Index Operation .....	70
18.3. Real Time Data Mirroring .....	70
18.4. Configuration .....	70
<b>19. iBatis</b> .....	
19.1. Introduction .....	72
19.2. Index Operation .....	72
19.3. Configuration .....	72
IV. Compass::Spring .....	
<b>20. Introduction</b> .....	
20.1. Overview .....	74

20.2. Compass Definition in Application Context .....	74
<b>21. DAO Support .....</b>	
21.1. Dao and Template .....	76
<b>22. Spring Transaction .....</b>	
22.1. Introduction .....	77
22.2. LocalTransaction .....	77
22.3. JTASyncTransaction .....	77
22.4. SpringSyncTransaction .....	77
22.5. CompassTransactionManager .....	77
<b>23. Hibernate 3 Gps Device Support .....</b>	
23.1. Introduction .....	78
23.2. SpringHibernate3GpsDevice .....	78
<b>24. OJB Gps Device Support .....</b>	
24.1. Introduction .....	79
24.2. SpringOjbGpsDevice .....	79
24.3. SpringOjbGpsDeviceInterceptor .....	79
<b>25. Jdbc Gps Device Support .....</b>	
25.1. Introduction .....	80
25.2. ResultSet Mapping .....	80
25.3. Table Mapping .....	82
<b>26. Spring AOP .....</b>	
26.1. Introduction .....	84
26.2. Advices .....	84
26.3. Dao Sample .....	84
26.4. Transactional Service Sample .....	85
<b>27. Spring MVC Support .....</b>	
27.1. Introduction .....	87
27.2. Support Classes .....	87
27.3. Index Controller .....	87
27.4. Search Controller .....	87
V. Compass::Samples .....	
<b>28. Library Sample .....</b>	
28.1. Introduction .....	89
28.2. Running The Sample .....	89
<b>29. Petclinic Sample .....</b>	
29.1. Introduction .....	90
29.2. Running The Sample .....	90
29.3. Data Model In Petclinic .....	90
29.3.1. Common Meta-data (Optional) .....	90
29.3.2. Resource Mapping .....	91
29.3.3. OSEM .....	92
29.4. Data Access In Petclinic .....	92
29.4.1. Hibernate .....	92
29.4.2. Apache OJB .....	92
29.4.3. JDBC .....	92
29.5. Web (MVC) in Petclinic .....	93
VI. Appendixes .....	
A. Configuration Settings .....	
A.1. Compass Configuration Settings .....	95
A.1.1. compass.engine.connection .....	95
A.1.2. JNDI .....	95
A.1.3. Property .....	96

---

A.1.4. Transaction Level .....	96
A.1.5. Transaction Strategy .....	97
A.1.6. Property Accessor .....	98
A.1.7. Converters .....	98
A.1.8. Search Engine .....	102
A.1.9. Search Engine Jdbc .....	104
A.1.10. Search Engine Analyzers .....	107
A.1.11. Search Engine Analyzer Filters .....	108
A.1.12. Search Engine Highlighters .....	109
A.1.13. Other Settings .....	110
B. Lucene Jdbc Directory .....	
B.1. Overview .....	112
B.2. Performance Notes .....	114
B.3. Transaction Management .....	114
B.3.1. Auto Commit Mode .....	114
B.3.2. DataSource Transaction Management .....	115
B.3.3. Using External Transaction Manager .....	115
B.3.4. DirectoryTemplate .....	116
B.4. File Entry Handler .....	116
B.4.1. IndexInput Types .....	117
B.4.2. IndexOutput Types .....	118

---

# Preface

Compass is a powerful, transactional Object to Search Engine Mapping (OSEM) Java framework. Compass allows you to declaratively map your Object domain model to the underlying Search Engine, synchronising data changes between Index and different datasources. Compass provides a high level abstraction on top of the Lucene low level API. Compass also implements fast index operations and optimization and introduces transaction capabilities to the Search Engine.

Compass aim is to provide the following:

- The simplest solution for enabling search capabilities within your application stack.
- Promote the use of Search Engine as a lightweight application datasource.
- Provide rich Search Engine semantics to find application data.
- Synchronize data changes between Search Engine and datasource.
- Write less code, find data quicker.



---

# Part I. Compass::Core

Compass::Core aim is to simplify the integration of Java applications with Search Engine.

The Compass::Core functionality is explained in detail in the following chapters.

- **Configuration:** describes how to configure and work with Compass and explains all of Compass's configuration parameters.
  - **Search Engine:** an abstraction built on top of the Lucene search engine. Adds features like two phase transaction management, fast updates and optimisers.
  - **OSEM:** Object Search Engine Mapping, provides the ability to map POJO's (Plain Old Java Objects) to the search engine through XML configuration files. Introduces the syntax for configuring Compass to work with Objects.
  - **Common-metadata:** a Compass feature for externalising the definition of meta-data names and aliases used in OSEM files. This provides the ability to centralise the management of meta-data.
  - **Resource Mapping:** provides the ability to map resources other than Objects for indexing within the search engine. Introduces the syntax for configuring Compass to work with Resources.
  - **Transaction:** describes Compass transaction support. Compass supports multiple transaction strategies and comes built in with LocalTransaction and JTA synchronisation support.
-

---

# Chapter 1. Introduction

## 1.1. Overview

Compass is built using a layered architecture. Applications interact with the underlying Search Engine through three main Compass interfaces: `Compass`, `CompassSession` and `CompassTransaction`. These interfaces hide the implementation details of the Compass Search Engine abstraction layer.

`Compass` provides access to search engine management functionality and `CompassSession`'s for managing data within the Search Engine. It is created using `CompassConfiguration` (loads configuration and mappings files). When `Compass` is created, it will either join an existing index or create a new one if none is available, then instantiates the configured Search Engine Optimizer. After this, an application will use `Compass` to obtain a `CompassSession` in order to start managing the data with the Search Engine. `Compass` is a heavyweight object, usually created at application startup and shared within an application for `CompassSession` creation.

`CompassSession` as the name suggests, represents a working lightweight session within `Compass::Core`. With a `CompassSession`, applications can save and retrieve any meta data (declared in `Compass` mapping files) from the Search Engine. Applications work with `CompassSession` at either the Object level or `Compass` Resource level to save and retrieve data. In order to work with Objects within `Compass::Core`, they must be specified within a Object/Search Engine Mapping (OSEM) file. In order to work with Resources, they must be specified within a Resource Mapping file. `Compass::Core` will then retrieve the declared meta data from the Object automatically when saving Objects within `Compass`. When querying the Search Engine, `Compass::Core` provides a `CompassHits` interface which one can use to work with the search engine results (getting scores, resources and mapped objects).

`CompassTransaction`, retrieved from the `CompassSession` and is used to manage transactions within `Compass`. You can configure `Compass::Core` to use either local transactions or JTA synchronization. Note, that unlike JDBC, automatic transaction registration will not happen, so we strongly recommend using the `CompassTransaction` abstraction for easy (configuration based) replacement of the transaction strategy.

After so many words, let's see a code snippet that shows the usage of the main compass interfaces:

```
CompassConfiguration conf =
    new CompassConfiguration().configure().addClass(Author.class);
Compass compass = conf.buildCompass();
CompassSession session = compass.openSession();
CompassTransaction tx = null;
try {
    tx = session.beginTransaction();
    ...
    session.save(author);
    CompassHits hits = session.find("jack london");
    Author a = (Author) hits.data(0);
    Resource r = hits.getResource(0);
    ...
    tx.commit();
} catch (CompassException ce) {
    if (tx != null) tx.rollback();
} finally {
    session.close();
}
```

## 1.2. Template and Callback

`Compass::Core` also provides a simple implementation of the template design pattern, using the

`CompassTemplate` and the `CompassCallback` classes. Using it, one does not have to worry about the `Compass` session or transaction handling. The `CompassTemplate` provides all the session operations, except that they are transactional (a new session is opened and a new transaction is created and committed when calling them). It also provides the `execute` method, which accepts a callback class (usually an anonymous inner class), to execute within it operations that are wrapped within the same transaction.

An example of using the template is provided:

```
CompassConfiguration conf =
    new CompassConfiguration().configure().addClass(Author.class);
Compass compass = conf.buildCompass();
CompassTemplate template = new CompassTemplate(compass);
template.save(author); // open a session, transaction, and closes both
Author a = (Author) template.execute(new CompassCallback() {
    public Object doInCompass(CompassSession session) {
        // all the actions here are within the same session
        // and transaction
        session.save(author);
        CompassHits hits = session.find("london");
        ...
        return session.load(id);
    }
});
```

---

# Chapter 2. Configuration

## 2.1. Introduction

Compass must be configured to work with a specific applications domain model. There are a large number of configuration parameters available (with default settings), which controls how Compass works internal and with the underlying Search Engine. This section describes the configuration API and parameters.

## 2.2. Programmatic Configuration

An instance of `CompassConfiguration` represents a set of mappings (one or more OSEM or Resource mappings), Common Meta Data definitions, transaction and Search Engine parameters. `CompassConfiguration` is used to build an immutable `Compass` instance.

`CompassConfiguration` provides several API's for adding OSEM and Resource mapping (suffixed `.cpm.xml`), as well as Common Meta Data definition (suffixed `.cmd.xml`). The following table summarizes the most important API's:

**Table 2.1. Configuration Mapping API**

API	Description
<code>addFile(String)</code>	Loads the mapping file ( <code>cpm</code> or <code>cmd</code> ) according to the specified file path string.
<code>addFile(File)</code>	Loads the mapping file ( <code>cpm</code> or <code>cmd</code> ) according to the specified file object reference.
<code>addClass(Class)</code>	Loads the mapping file ( <code>cpm</code> ) according to the specified class. <code>test.Author.class</code> will map to <code>test/Author.cpm.xml</code> within the class path.
<code>addURL(URL)</code>	Loads the mapping file ( <code>cpm</code> or <code>cmd</code> ) according to the specified URL.
<code>addResource(String)</code>	Loads the mapping file ( <code>cpm</code> or <code>cmd</code> ) according to the specified resource path from the class path.
<code>addInputStream(InputStream)</code>	Loads the mapping file ( <code>cpm</code> or <code>cmd</code> ) according to the specified input stream.
<code>addDirectory(String)</code>	Loads all the files named <code>*.cpm.xml</code> or <code>*.cmd.xml</code> from within the specified directory.
<code>addJar(File)</code>	Loads all the files named <code>*.cpm.xml</code> or <code>*.cmd.xml</code> from within the specified Jar file.
<code>addMappingResolver(MappingResolver)</code>	Uses a class that implements the <code>MappingResolver</code> to get an <code>InputStream</code> for xml mapping definitions.

Other than mapping file configuration API (`CompassConfiguration`), `Compass::Core` can be configured through the `CompassSettings` interface. `CompassSettings` is similar to `Java Properties` class and is accessible

via the `CompassConfiguration.getSettings()` or the `CopmassConfiguration.setSetting(String setting, String value)` methods. Compass's many different settings are explained in the Configuration Settings appendix.

Compass setting can also be defined programmatically using the `org.compass.core.config.CompassEnvironment` and `org.compass.core.lucene.LuceneEnvironment` classes (hold programmatic manifestation of all the different settings).

In terms of required settings, Compass only requires the `compass.engine.connection` (which maps to `CompassEnvironment.CONNECTION`) parameter defined.

Global Converters (classes that implement `Compass Converter`) can also be registered with the configuration to be used by the created compass instances. The `Converters` are registered under a logical name, and can be referenced in the mapping definitions. The method to register a global converter is `registerConverter`.

Again, many words and so little code... . The following code example shows the minimal `CompassConfiguration` with programmatic control:

```
CompassConfiguration conf = new CompassConfiguration()
    .setSetting(CompassEnvironment.CONNECTION, "my/index/dir")
    .addResource(DublinCore.cmd.xml)
    .addClass(Author.class);
```

## 2.3. XML Configuration

All of Compass's operational configuration (apart from mapping definitions) can be defined in a single xml configuration file, with the default name `compass.cfg.xml`. You can define the environmental settings and mapping file locations within this file. The following table shows the different `CompassConfiguration` API's for locating the main configuration file:

**Table 2.2. Compass Configuration API**

API	Description
<code>configure()</code>	Loads a configuration file called <code>compass.cfg.xml</code> from the root of the class path.
<code>configure(String)</code>	Loads a configuration file from the specified path

### 2.3.1. Schema Based Configuration

The preferred way to configure Compass (and the simplest way) is to use an Xml configuration file, which validates against a Schema. It allows for richer and more descriptive (and less erroneous) configuration of Compass. The schema is fully annotated, with each element, attribute documented within the schema. Note, that some additional information is explained in the Configuration Settings appendix, even if it does not apply in terms of the name of the setting to be used, it is advisable to read the appropriate section for more fuller explanation (such as converters, highlighters, analyzers, and so on).

Here are a few sample configuration files, the first is a simple file based index with the OSEM definitions for the Author class.

```
<compass-core-config xmlns="http://www.opensymphony.com/compass/schema/core-config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opensymphony.com/compass/schema/core-config
```

```

    http://www.opensymphony.com/compass/schema/compass-core-config.xsd">

<compass name="default">
  <connection>
    <file path="target/test-index"/>
  </connection>

  <mappings>
    <class name="test.Author" />
  </mappings>

</compass>
</compass-core-config>

```

The next sample configures a jdbc based index, with a bigger buffer size for default file entries:

```

<compass-core-config xmlns="http://www.opensymphony.com/compass/schema/core-config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/compass/schema/core-config
    http://www.opensymphony.com/compass/schema/compass-core-config.xsd">

  <compass name="default">

    <connection>
      <jdbc dialect="org.apache.lucene.store.jdbc.dialect.HSQLDialect">
        <dataSourceProvider>
          <driverManager url="jdbc:hsqldb:mem:test" username="sa" password=""
            driverClass="org.hsqldb.jdbcDriver" />
        </dataSourceProvider>
        <fileEntries>
          <fileEntry name="__default__">
            <indexInput bufferSize="4096" />
            <indexOutput bufferSize="4096" />
          </fileEntry>
        </fileEntries>
      </jdbc>
    </connection>
  </compass>
</compass-core-config>

```

The next sample configures a jdbc based index, with a JTA transaction (note the managed="true" and commitBeforeCompletion="true"):

```

<compass-core-config xmlns="http://www.opensymphony.com/compass/schema/core-config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/compass/schema/core-config
    http://www.opensymphony.com/compass/schema/compass-core-config.xsd">

  <compass name="default">

    <connection>
      <jdbc dialect="org.apache.lucene.store.jdbc.dialect.HSQLDialect" managed="true">
        <dataSourceProvider>
          <driverManager url="jdbc:hsqldb:mem:test" username="sa" password=""
            driverClass="org.hsqldb.jdbcDriver" />
        </dataSourceProvider>
      </jdbc>
    </connection>
    <transaction factory="org.compass.core.transaction.JTASyncTransactionFactory" commitBeforeCompletion="true">
    </transaction>
  </compass>
</compass-core-config>

```

Here is another sample, that configures another analyzer, a specialized Converter, and changed the default date format for all Java Dates (date is an internal name that maps to Compass default date Converter).

```

<compass-core-config xmlns="http://www.opensymphony.com/compass/schema/core-config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opensymphony.com/compass/schema/core-config
    http://www.opensymphony.com/compass/schema/compass-core-config.xsd">

```

```

<compass name="default">

  <connection>
    <file path="target/test-index"/>
  </connection>

  <converters>
    <converter name="date" type="org.compass.core.converter.basic.DateConverter">
      <setting name="format" value="yyyy-MM-dd" />
    </converter>
    <converter name="myConverter" type="test.Myconverter" />
  </converters>

  <searchEngine>
    <analyzer name="test" type="Snowball" snowballType="Lovins">
      <stopWords>
        <stopWord value="test" />
      </stopWords>
    </analyzer>
  </searchEngine>
</compass>
</compass-core-config>

```

The next configuration uses `batch_insert` transaction, with a higher max buffered documents for faster batch indexing.

```

<compass-core-config xmlns="http://www.opensymphony.com/compass/schema/core-config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.opensymphony.com/compass/schema/core-config
http://www.opensymphony.com/compass/schema/compass-core-config.xsd">

  <compass name="default">

    <connection>
      <file path="target/test-index"/>
    </connection>

    <transaction isolation="batch_insert">
      <batchInsertSettings maxBufferedDocs="100" />
    </transaction>
  </compass>
</compass-core-config>

```

### 2.3.2. DTD Based Configuration

Compass can be configured using a DTD based xml configuration. The DTD configuration is less expressive than the schema based one, allowing to configure mappings and Compass settings. The Configuration Settings are explained in an appendix.

And here is an example of the xml configuration file:

```

<!DOCTYPE compass-core-configuration PUBLIC
"-//Compass/Compass Core Configuration DTD 1.0//EN"
"http://www.opensymphony.com/compass/dtd/compass-core-configuration.dtd">

<compass-core-configuration>
  <compass>
    <setting name="compass.engine.connection">my/index/dir</setting>

    <meta-data resource="vocabulary/DublinCore.cmd.xml" />
    <mapping resource="test/Author.cpm.xml" />

  </compass>
</compass-core-configuration>

```

## 2.4. Obtaining a Compass reference

After `CompassConfiguration` has been set (either programmatic or using the XML configuration file), you can now build a `Compass` instance. `Compass` is intended to be shared among different application threads. The following simple code example shows how to obtain a `Compass` reference.

```
Compass compass = cfg.buildCompass();
```

Note: It is possible to have multiple `Compass` instances within the same application, each with a different configuration.

---

# Chapter 3. Search Engine

## 3.1. Introduction

Compass::Core provides an abstraction layer on top of the wonderful [Lucene](#) Search Engine. Compass::Core also provides several additional features on top of Lucene, like two phase transaction management, fast updates, and optimizers. When trying to explain how Compass::Core works with the Search Engine, first we need to understand the Search Engine domain model.

## 3.2. Alias, Resource and Property

`Resource` represents a collection of properties. You can think about it as a virtual document - a chunk of data, such as a web page, an email message, or a serialization of the Author object. A `Resource` is always associated with a single `Alias` and several `Resources` can have the same `Alias`. A `Property` is just a place holder for a name and value (both strings). A `Property` within a `Resource` represents some kind of meta-data that is associated with the `Resource` like the author name. In data-base terms, you can think of an `Alias` as a table, the `Resource` as a row in the table and `Property` as the column (with a value). Note: a `Resource` can have several properties with the same name.

Every `Resource` is associated with one or more id properties. They are required for Compass::Core to manage `Resource` loading based on ids and `Resource` updates (a well known difficulty when using Lucene directly). Id properties are defined either explicitly in the Resource Mapping definition or implicitly in the OSEM definition.

For Lucene users, Compass `Resource` maps to Lucene `Document` and Compass `Property` maps to Lucene `Field`.

## 3.3. Creating Resource and Property

In order to create a `Resource` and a `Property`, you use the `CompassSession` which acts as a factory. `CompassSession` provides several API's:

- `createResource(String alias)`: Creates a `Resource` with the specified alias.
- `createProperty(String name, String value, Property.Store store, Property.Index index)`: Creates a `Property` with the specified name and value. As well as the `Property` behavioural aspect within the Search Engine.
- `createProperty(String name, String value, Property.Store store, Property.Index index, Property.TermVector termVector)`: Creates a `Property` with the specified name and value. As well as the `Property` behavioural aspect within the Search Engine.

When creating a `Property`, you must specify the `store` and the `index` parameters.

Another option when creating `Resource` `Property` is to define resource mapping and within it a `resource-id` and `resource-property` mappings (please see the Resource Mapping Section). When defining the mappings, Compass can be smart enough to guess the type, index, store, and other options using the mappings, allowing the usage of the simple `addProeprty(String propertyName, Object value)` API of `Resource` (even auto

converting the Object to the correct value using Compass converter architecture).

The following table specifies the available values for the `store` parameter:

**Table 3.1.**

Store	Description
<code>Property.Store.NO</code>	Do not store the property value in the index (won't be able to retrieve it later on).
<code>Property.Store.YES</code>	Stores the original property value in the index.
<code>Property.Store.COMPRESS</code>	Stores the original property value in the index in a compressed form.

The following table specifies the available values for the `index` parameter:

**Table 3.2.**

Index	Description
<code>Property.Index.NO</code>	Do not index the property value. This property can thus not be searched, but one can still access its contents provided it is stored.
<code>Property.Index.TOKENIZED</code>	Index the property value so it can be searched. An Analyzer will be used to tokenize and possibly further normalize the text before its terms will be stored in the index.
<code>Property.Index.UN_TOKENIZED</code>	Index the property value without using an Analyzer, so it can be searched. As no analyzer is used, the value will be stored as a single term (perfect for id like properties).

The following table specifies the available values for the `termVector` parameter:

**Table 3.3.**

Term Vector	Description
<code>Property.TermVector.NO</code>	Do not store any term vector information (the default behavior).
<code>Property.TermVector.YES</code>	Store the term vectors of each document. A term vector is a list of the resources's terms and their number of occurrences in that document.
<code>Property.TermVector.WITH_POSITIONS</code>	Store the term vector + Token offset information.
<code>Property.TermVector.WITH_OFFSETS</code>	Store the term vector + Token offset information.
<code>Property.TermVector.WITH_POSITIONS_OFFSETS</code>	Store the term vector + Token position and offset information.

The following code shows how you can create a `Resource` with `Compass::Core` and save it.

```
CompassSession session = compass.openSession();
CompassTransaction tx = session.beginTransaction();
Resource authorResource = session.createResource("author");
Property authorIdProp = session.createProperty("id", "AUTHOR0812",
    Property.Store.YES, Property.Index.UN_TOKENIZED);
Property authorNameProp = session.createProperty("name",
    "Jack London", Property.Store.YES, Property.Index.TOKENIZED);
authorResource.addProperty(authorIdProp);
authorResource.addProperty(authorNameProp);
session.save(resource);
tx.commit();
```

### 3.3.1. Boosting Resource and Property

`Compass::Core` allows you to set the boosting factor for `Resource` and `Property` (through Lucene boosting feature). Boosting is the process of making a `Resource` or a `Property` more or less "important" than others.

Initially, `Resource` and `Property` have no boost (actually, a boost of 1.0). You can set the boost level on a `Resource` (which propagates to all the properties that have no boosting set) or on a specific `Property`. Higher values than 1.0 makes it more relevant and values lower than 1.0 make it less relevant.

## 3.4. Analyzers

`Analyzers` are components that pre-process input text. They are also used when searching (the search string has to be processed the same way that the indexed text was processed). Therefore, it is important to use the same `Analyzer` for both indexing and searching.

`Compass::Core` can be configured to have multiple analyzers, registered under different analyzer names. It has two internal analyzers: `default` and `search`, as defined in the `Analyzers` section in the `Configuration` chapter.

`Analyzer` is a Lucene class (which qualifies to `org.apache.lucene.analysis.Analyzer` class). Lucene core itself comes with several `Analyzers` and you can configure `Compass::Core` to work with either one of them. If we take the following sentence: "The quick brown fox jumped over the lazy dogs", we can see how the different `Analyzers` handle it:

```
whitespace (org.apache.lucene.analysis.WhitespaceAnalyzer)
```

```
[The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]
```

```
simple (org.apache.lucene.analysis.SimpleAnalyzer)
```

```
[the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]
```

```
stop (org.apache.lucene.analysis.StopAnalyzer)
```

```
[quick] [brown] [fox] [jumped] [over] [lazy] [dogs]
```

```
standard (org.apache.lucene.analysis.standard.StandardAnalyzer)
```

```
[quick] [brown] [fox] [jumped] [over] [lazy] [dogs]
```

`Analyzers` have a list of stop words which they exclude while analyzing, you can control both the `Analyzer` and the stop words using `Compass::Core` configuration parameters.



Compass::Core, every transaction that performed one or more `save` or `create` operation, and committed successfully, creates another segment in the respective index (different than how Lucene manages it's index), which helps in implementing quick transaction commits, as well as paving the way for a two phase commit support (and the reason behind having optimizers).

### 3.6.3. serializable

The `serializable` transaction level operates the same as the `read_committed` transaction level, except that when the transaction is opened/started, a lock is acquired on all the sub-indexes. This causes the transactional operations to be sequential in nature (as well as being a performance killer).

### 3.6.4. batch\_insert

A special transaction level, `batch_insert` utilizes the extremely fast batch indexing provided by Lucene. The transaction supports only `create` operation, but note that if another `Resource` with the same alias and ids already exists in the system, you will have two instances of it in the index (in other words, `create` doesn't delete the old `Resource`). You can control the `batch_insert` transaction using several settings which are explained in the Configuration section. An important note is that the transaction is not a transaction which can be rolled back, since Lucene commits the changes during the batch indexing process, which means that a `rollback` operation won't rollback the changes. The index is optimized when the transaction is committed, which means that all the segments are merged to one segment, in order to provide fast searching. The transaction is mainly used for background batch indexing.

## 3.7. Optimizers

As mentioned in the `read_committed` section, every dirty transaction that is committed successfully creates another segment in the respective index. The more segments the index has, the slower the fetching operations take. That's why it is important to keep the index optimized and with a controlled number of segments. We do this by merging small segments into larger segments.

In order to solve the problem, Compass::Core has a `SearchEngineOptimizer` which is responsible for keeping the number of segments at bay. When Compass is built using `CompassConfiguration`, the `SearchEngineOptimizer` is started and when the Compass is closed, the `SearchEngineOptimizer` is stopped.

### 3.7.1. Scheduled Optimizers

Compass::Core provides support for scheduled optimizers. The scheduled optimizers uses `Java Timer` to control it's execution. `SearchEngineOptimizer` starts and stops the timer when it starts and stops. There are several settings parameters that can be set to control the scheduling.

Note: each optimizer that Compass provides can be scheduled.

### 3.7.2. Aggressive Optimizer

The `AggressiveOptimizer` uses Lucene optimization feature to optimize the index. Lucene optimization merges all the segments into one segment. You can set the limit of the number of segments, after which the index is considered to need optimization (the aggressive optimizer merge factor).

### 3.7.3. Adaptive Optimizer

The `AdaptiveOptimizer` uses optimize the segments while trying to manage the optimization time at bay. As an example, when we have a large segment in our index (for example, after we batched indexed the data), and we perform several interactive transactions, the aggressive optimizer will than merge all the segments together, while the adaptive optimizer will only merge the new small segments. You can set the limit of the number of segments, after which the index is considered to need optimization (the adaptive optimizer merge factor).

### 3.7.4. Null Optimizer

`Compass::Core` also comes with a `NullOptimizer`, which performs no optimizations. It is mainly there if the hosting application developed it's own optimization which is maintained by other means than the `SearchEngineOptimizer`. It also makes sense to use it when configuring a `Compass` instance with a `batch_insert` transaction.

Note that when using the `NullOptimizer` it makes no sense to use the scheduling feature, so remember to set the `compass.engine.optimizer.schedule` to `false`.

---

# Chapter 4. OSEM - XML

## 4.1. Introduction

Compass::Core provides the ability to map Java Objects to the underlying Search Engine through simple XML mapping files, we call this technology OSEM (Object Search Engine Mapping). OSEM provides a rich syntax for describing Object attributes and relationships. The OSEM files are used by Compass to extract the required property from the Object model at run-time and inserting the required meta-data into the Search Engine index.

## 4.2. Searchable Classes

Searchable classes are normally classes representing the state of the application, implementing the entities with the business model. Compass works best if the classes follow the simple Plain Old Java Object (POJO) programming model. The following class is an example of a searchable class:

```
import java.util.Date;
import java.util.Set;

public class Author {
    private Long id; // identifier
    private String name;
    private Date birthday;
    private Set books;

    private void setId(Long id) {
        this.id = id;
    }

    public Long getId() {
        return this.id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public Date getBirthday() {
        return this.birthday;
    }

    public void setBooks(Set books) {
        this.books = books;
    }

    public Set getBooks() {
        return this.books;
    }

    // addBook not needed by Compass::Core
    public void addBook(Book book) {
        this.books.add(book);
    }
}
```

Compass works non-intrusive with application Objects, these Objects must follow several rules:

### 4.2.1. Implement a Default Constructor

`Author` has an implicit default (no-argument) constructor. All persistent classes must have a default constructor (which may be non-public) so `Compass::Core` can instantiate using `Constructor.newInstance()`.

### 4.2.2. Provide Property Identifier(s)

OSEM requires that any mapped Object will define one or more properties (JavaBean properties) that identifies the class. The id properties can be called anything, and it's type can be any primitive type, primitive "wrapper" type, `java.lang.String` or `java.util.Date`.

### 4.2.3. Declare Accessors and Mutators (Optional)

Even though `Compass` can directly persist instance variables, it is usually better to decouple this implementation detail from the Search Engine mechanism. `Compass::Core` recognizes JavaBean style property (`getFoo`, `isFoo`, and `setFoo`). This mechanism works with any level of visibility.

### 4.2.4. Implementing equals() and hashCode()

You have to override the `equals()` and `hashCode()` methods if you intend to mix objects of persistent classes (e.g. in a `Set`). You can implement it by using the identifier of both objects, but note that `Compass::Core` works best with surrogate identifier (and will provide a way to automatically generate them), thus it is best to implement the methods using business keys.

## 4.3. Mapping

Object/Search Engine mappings are defined in an XML document. The mapping language is Java centric, meaning that mappings are constructed around the classes themselves and not internal `Resources`. A possible OSEM file for the previous `Author` class example follows:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
  "-//Compass/Compass Core Mapping DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-mapping.dtd">

<compass-core-mapping package="eg">

  <class name="Author" alias="author">

    <id name="id" />

    <constant>
      <meta-data>type</meta-data>
      <meta-data-value>person</meta-data-value>
      <meta-data-value>author</meta-data-value>
    </constant>

    <property name="name">
      <meta-data>name</meta-data>
      <meta-data>authorName</meta-data>
    </property>

    <property name="birthday">
      <meta-data>birthday</meta-data>
    </property>

    <component name="books" ref-alias="book" />

    <!-- can be a reference instead of component
    <reference name="books" ref-alias="book" />
```

```

-->
</class>

<class name="Book" alias="book">

    ...

</class>

</compass-core-mapping>

```

The above example defines the mapping for `Author` and `Book` classes. It introduces some key Compass mapping concepts and syntax. Before explaining the concepts, it is essential that the terminology used is clearly understood.

The first issue to address is the usage of the term `Property`. Because of its common usage as a concept in Java and Compass (to express Search Engine and Semantic terminology), special care has been taken to clearly prefix the meaning. A class `Property` refers to a Java class attribute. A `Resource Property` refers in Compass to Search Engine meta-data, which contains the values of the mapped class `Property` value. In previous OSEM example, the value of class `Property` "name" is mapped to two `Resource Property` instances called "name" and "authorname", each containing the value of the class `Property` "name".

The OSEM example above shows:

- The unique class identifier, which maps to the "id" class property.
- Constant meta data, a feature that allows Compass to insert extra meta data and values (not expressed in the Object). Compass::Core will save the `Resource Property` "type" with the specified values "person" and "author".
- The mappings for the class `Property` "name" saved with two `Resource Property` called "name" and "authorName".
- A dependency between `Author` and `Book` managed using a `component mapping`.

Each of these concepts are explained in detail in the following sections.

All XML mappings should declare the doctype shown. The actual DTD may be found at the URL above, or in the `compass-core-x.x.x.jar`. Compass will always look for the DTD in the classpath first.

### 4.3.1. compass-core-mapping

The main element which holds all the rest of the mappings definitions.

```

<compass-core-mapping package="packageName" />

```

**Table 4.1.**

Attribute	Description
package (optional)	Specifies a package prefix for unqualified class names in the mapping document.

### 4.3.2. class

Declaring a searchable class using the `class` element.

```
<class
  name="className"
  alias="alias"
  sub-index="sub index name"
  analyzer="name of the analyzer"
  root="true|false"
  poly="false|true"
  poly-class="the class name that will be used to instantiate poly mapping (optional)"
  extends="a comma separated list of aliases to extend"
  boost="boost value for the class"
  all="true|false"
  all-term-vector="no|yes|positions|offsets|positios_offsets"
  all-metadata="all meta-data"
  all-analyzer="name of the analyzer used for the all property"
  converter="converter lookup name"
>
  (id)*,
  parent?,
  (analyzer?),
  (property|component|reference|constant)*
</class>
```

**Table 4.2.**

Attribute	Description
name	The fully qualified class name (or relative if the package is declared in <code>compass-core-mapping</code> ).
alias	The alias of the <code>Resource</code> that will be mapped to the class.
sub-index (optional, defaults to the alias value)	The name of the sub-index that the alias will map to. When joining several searchable classes into the same index, the search will be much faster, but updates perform locks on the sub index level, so it might slow it down.
analyzer (optional, defaults to the default analyzer)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> properties. Defaults to the <code>default</code> analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the <code>analyzer</code> mapping (a child mapping of class mapping) (for a property value that controls the analyzer), the <code>analyzer</code> attribute will have no effects.
root (optional, defaults to <code>true</code> )	Specifies if the class is a "root" class or not. You should define the searchable class with <code>false</code> if it only acts as mapping definitions for a component mapping.
poly (optional, defaults to <code>false</code> )	Specifies if the class will be enabled to support polymorphism. This is the less preferable way to map an inheritance tree, since the <code>extends</code> attribute can be used to statically extend base classes or contracts.
poly-class (optional)	If <code>poly</code> is set to <code>true</code> , the actual class name of the indexed object will be saved to the index as well (will be used later to instantiate the Object). If the <code>poly-class</code> is set, the class name will not be saved to the index, and the value of <code>poly-class</code> will be used to instantiate

Attribute	Description
	all the classes in the inheritance tree.
extends (optional)	A comma separated list of aliases to extend. Can extend a <code>class</code> mapping or a <code>contract</code> mapping. Note that can extend more than one <code>class/contract</code>
boost (optional, defaults to 1.0)	Specifies the boost level for the class.
all (optional, defaults to <code>true</code> )	Controls if the searchable class will create it's own internal "all" meta-data. The "all" meta-data holds searchable information of all the class searchable content.
all-term-vector (optional, defaults to configuration setting <code>compass.property.all.termVector</code> )	The term vector value of the all property.
all-metadata (optional, defaults to configuration setting <code>compass.property.all</code> )	The name of the all property.
all-analyzer (optional, defaults to configuration setting <code>compass.engine.all.analyzer</code> , which in turn, defaults to the default analyzer)	The name of the analyzer that will be used to analyze the all property.
converter (optional)	The global converter lookup name registered with the configuration. Responsible for converting the <code>ClassMapping</code> definition. Defaults to compass internal <code>ClassMappingConverter</code> .

Root classes have their own index within the search engine index directory. Classes with a dependency to Root class, that don't require an index (i.e. component) should set `root` to `false`. You can control the sub-index that the root classes will map to using the `sub-index` attribute, otherwise it will create a sub-index based on the alias name.

If the class can be mapped to several classes (i.e. it is an interface or an abstract class), than set `poly` to `true`. This means Compass will persist the fully qualified class in the index.

You can set the boost level at the class level, which is applied to all class meta data (override by specifying at meta data level).

The `class` mapping can extend other `class` mappings (more than one), as well as `contract` mappings. All the mappings that are defined within the `class` mapping or the `contract` mapping will be inherited from the extended mappings. You can add any defined mappings by defining the same mappings in the `class` mappings, except for id mappings, which will be overridden. Note that any `xml` attributes (like `root`, `sub-index`, ...) that are defined within the extended mappings are not inherited.

The default behavior of the searchable class will support the "all" feature, which means that compass will create an "all" meta-data which represents all the other meta-data (with several exceptions, like `Reader` class property). The name of the "all" meta-data will default to the compass setting, but you can also set it using the `all-metadata` attribute.

### 4.3.3. contract

Declaring a searchable contract using the `contract` element.

```
<contract
  alias="alias"
>
  (id)*,
  (analyzer?),
  (property|component|reference|constant)*
</contract>
```

**Table 4.3.**

Attribute	Description
alias	The alias of the contract. Will be used as the alias name in the <code>class</code> mapping extended attribute

A contract acts as an interface in the Java language. You can define the same mappings within it that you can define in the `class` mapping, without defining the class that it will map to.

If you have several classes that have similar properties, you can define a `contract` that joins the properties definition, and then extend the contract within the mapped classes (even if you don't have a concrete interface or class in your Java definition).

### 4.3.4. id

Declaring a searchable id class property (a.k.a JavaBean property) of a class using the `id` element.

```
<id
  name="property name"
  accessor="property|field"
  boost="boost value for the class property"
  class="explicit declaration of the property class"
  managed-id="auto|true|false"
  exclude-from-all="false|true"
  converter="converter lookup name"
>
  (meta-data)*
</id>
```

**Table 4.4.**

Attribute	Description
name	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
accessor (optional, defaults to <code>property</code> )	The strategy to access the class property value. <code>property</code> access using the Java Bean accessor methods, while <code>field</code> directly access the class fields.
boost (optional, default to <code>1.0f</code> )	The boost level that will be propagated to all the meta-data defined within the <code>id</code> .
class (optional)	An explicit definition of the class of the property, helps for certain converters.
managed-id (optional, defaults to <code>auto</code> )	The strategy for creating or using a class property meta-data id

Attribute	Description
	(which maps to a <code>ResourceProperty</code> ).
exclude-from-all (optional, defaults to false)	Excludes the class property from participating in the "all" meta-data, unless specified in the meta-data level.
converter (optional)	The global converter lookup name registered with the configuration.

The id mapping is used to map the class property that identifies the class. You can define several id properties, even though we recommend using one. You can use the id mapping for all the Java primitive types (i.e. `int`), Java primitive wrapper types (i.e. `Integer`) and the `String` type.

Compass::Core requires that `id` and `property` mappings will be identifiable on the root class (`Resource`) level. Compass does that by either using one of the meta-data names (which is unique within ALL of the meta-data in the class mapping), or creating an internal one. Compass will create an internal one if no meta-data is defined in the `id` or `property` mapping. You can control it by using the `managed-id`, the value `auto` leaves the id assignment / creation as Compass's responsibility. Compass will analyze all the different meta-data defined in the mappings and will decide if it needs to create an internal id for an `id` or a `property` mapping. The `true` option will always create an internal id for the `id` or `property` and the `false` option will always take the first meta-data and use it as the `id` or `property` id.

### 4.3.5. property

Declaring a searchable class property (a.k.a JavaBean property) of a class using the `property` element.

```
<property
  name="property name"
  accessor="property|field"
  boost="boost value for the property"
  class="explicit declaration of the property class"
  analyzer="name of the analyzer"
  managed-id="auto|true|false"
  managed-id="[compass.managedId.index setting]|no|un_tokenized"
  exclude-from-all="false|true"
  converter="converter lookup name"
>
  (meta-data)*
</property>
```

Table 4.5.

Attribute	Description
name	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
accessor (optional, defaults to property)	The strategy to access the class property value. <code>property</code> means accessing using the Java Bean accessor methods, while <code>field</code> directly accesses the class fields.
boost (optional, default to 1.0f)	The boost level that will be propagated to all the meta-data defined within the class property.
class (optional)	An explicit definition of the class of the property, helps for certain converters (especially for <code>java.util.Collection</code> type properties, since it applies to the collection elements).

Attribute	Description
analyzer (optional, defaults to the class mapping analyzer decision scheme)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> meta-data mappings defined for the given property. Defaults to the class mapping analyzer decision scheme based on the analyzer set, or the <code>analyzer</code> mapping property.
override (optional, defaults to <code>true</code> )	If there is another definition with the same mapping name, if it will be overridden or added as additional mapping. Mainly used to override definitions made in extended mappings.
managed-id (optional, defaults to <code>auto</code> )	The strategy for creating or using a class property meta-data id (which maps to a <code>ResourceProperty</code> ).
managed-id-index (optional, defaults to <code>compass.managedId.index</code> setting, which defaults to <code>no</code> )	Can be either <code>un_tokenized</code> or <code>no</code> . It is the index setting that will be used when creating an internal managed id for a class property mapping (if it is not a property id, if it is, it will always be <code>un_tokenized</code> ).
exclude-from-all (optional, defaults to <code>false</code> )	Excludes the class property from participating in the "all" meta-data, unless specified in the meta-data level.
converter (optional)	The global converter lookup name registered with the configuration.

Compass::Core maps a class property to a set of meta-data (`Resource Property`).

You can map all internal Java primitive data types, primitive wrappers and most of the common Java classes (i.e. `Date` and `Calendar`). You can also map Arrays and Collections of these data types. When mapping a `Collection`, you must specify the object class (like `java.lang.String`) in the class mapping property.

The same rules for `managed-id` that apply for the `id` mapping, also applies for `property` mappings.

Note, that you can define a property with no `meta-data` mapping within it. It means that it will not be searchable, but the property value will be stored when persisting the object to the search engine, and it will be loaded from it as well (unless it is of type `java.io.Reader`).

### 4.3.6. analyzer

Declaring an analyzer controller property (a.k.a JavaBean property) of a class using the `analyzer` element.

```
<analyzer
  name="property name"
  null-analyzer="analyzer name if value is null"
  accessor="property|field"
  converter="converter lookup name"
>
</analyzer>
```

**Table 4.6.**

Attribute	Description
name	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
accessor (optional, defaults to	The strategy to access the class property value. <code>property</code> means

Attribute	Description
)	accessing using the Java Bean accessor methods, while <code>field</code> directly accesses the class fields.
<code>null-analyzer</code> (optional, defaults to <code>error</code> in case of a <code>null</code> value)	The name of the analyzer that will be used if the property has the <code>null</code> value.
<code>converter</code> (optional)	The global converter lookup name registered with the configuration.

The analyzer class property mapping, controls the analyzer that will be used when indexing the class data (the underlying `Resource`). If the mapping is defined, it will override the class mapping analyzer attribute setting.

If, for example, Compass is configured to have two additional analyzers, called `an1` (and have settings in the form of `compass.engine.analyzer.an1.*`), and another called `an2`. The values that the class property can hold are: `default` (which is an internal Compass analyzer, that can be configured as well), `an1` and `an2`. If the analyzer will have a `null` value, and it is applicable with the application, a `null-analyzer` can be configured that will be used in that case. If the class property has a value, but there is not matching analyzer, an exception will be thrown.

### 4.3.7. meta-data

Declaring and using the `meta-data` element.

```
<meta-data
  store="yes|no|compress"
  index="tokenized|un_tokenized|no"
  boost="boost value for the meta-data"
  analyzer="name of the analyzer"
  reverse="no|reader|string"
  exclude-from-all="[parent's exclude-from-all]|false|true"
  converter="converter lookup name"
  format="the format string (only applies to formatted elements)"
>
</meta-data>
```

**Table 4.7.**

Attribute	Description
<code>store</code> (optional, defaults to <code>yes</code> )	If the value of the class property that the meta-data maps to, is going to be stored in the index.
<code>index</code> (optional, defaults to <code>tokenized</code> )	If the value of the class property that the meta-data maps to, is going to be indexed (searchable). If it does, than controls if the value is going to be broken down and analysed ( <code>tokenized</code> ), or is going to be used as is ( <code>un_tokenized</code> ).
<code>boost</code> (optional, defaults to <code>1.0f</code> )	Controls the boost level for the <code>meta-data</code> .
<code>analyzer</code> (optional, defaults to the parent analyzer)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> meta-data. Defaults to the parent property mapping, which in turn defaults to the class mapping analyzer decision scheme based on the analyzer set, or the <code>analyzer</code> mapping property.
<code>reverse</code> (optional, defaults to <code>no</code> )	The meta-data will have it's value reversed. Can have the values of <code>no</code> - no reverse will happen, <code>string</code> - the reverse will happen and

Attribute	Description
	the value stored will be a reversed string, and <code>reader</code> - a special reader will wrap the string and reverse it. The <code>reader</code> option is more perform ant, but the <code>store</code> and <code>index</code> settings will be discarded.
<code>exclude-from-all</code> (optional, defaults to the parent's <code>exclude-from-all</code> value)	Excludes the meta-data from participating in the "all" meta-data.
<code>converter</code> (optional)	The global converter lookup name registered with the configuration. Note, that in case of a <code>Collection</code> property, the converter will be applied to the collection elements (Compass has it's own converter for Collections).
<code>format</code> (optional)	Allows for quickly setting a format for format-able types (dates, and numbers), without creating/registering a specialized converter under a lookup name.

The element `meta-data` is a `Property` within a `Resource`.

You can control the format of the marshalled values when mapping a `java.lang.Number` (or the equivalent primitive value) using the format provided by the `java.text.DecimalFormat`. You can also format a `java.util.Date` using the format provided by `java.text.SimpleDateFormat`. You set the format string in the `format` attribute.

### 4.3.8. component

Declaring and using the `component` element.

```
<component
  name="the class property name"
  ref-alias="name of the alias"
  max-depth="the depth of cyclic component mappings allowed"
  accessor="property|field"
  converter="converter lookup name"
>
</component>
```

**Table 4.8.**

Attribute	Description
<code>name</code>	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
<code>ref-alias</code> (optional)	The class mapping alias that defines the component. This is an optional attribute since under some conditions, compass can infer the correct reference alias.
<code>max-depth</code> (optional, defaults to 5)	The depth of cyclic component mappings allowed.
<code>override</code> (optional, defaults to <code>true</code> )	If there is another definition with the same mapping name, if it will be overridden or added as additional mapping. Mainly used to override definitions made in extended mappings.
<code>accessor</code> (optional, defaults to <code>property</code> )	The strategy to access the class property value. <code>property</code> access

Attribute	Description
)	using the Java Bean accessor methods, while <code>field</code> directly access the class fields.
converter (optional)	The global converter lookup name registered with the configuration.

The component element defines a class dependency within the root class. The dependency name is identified by the `ref-alias`, which can be non-rootable or have no `id` mappings.

An embedded class means that all the mappings (meta-data values) defined in the referenced class are stored within the alias of the root class. It means that a search that will hit one of the component mapped meta-datas, will return it's owning class.

The type of the JavaBean property can be the class mapping class itself, an `Array` or `Collection`.

Support for cyclic mapping (from one component to it's parent class) is implemented using the `parent` mapping.

### 4.3.9. reference

Declaring and using the `reference` element.

```
<reference
  name="the class property name"
  ref-alias="name of the alias"
  ref-comp-alias="name of an optional alias mapped as component"
  accessor="property|field"
  converter="converter lookup name"
>
</reference>
```

**Table 4.9.**

Attribute	Description
name	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
ref-alias (optional)	The class mapping alias that defines the reference. This is an optional attribute since under some conditions, compass can infer the correct reference alias.
ref-comp-alias (optional)	The class mapping alias that defines a "shadow component". Will marshal a component like mapping based on the alias into the current class. Note, it's best to create a dedicated class mapping (with <code>root="false"</code> ) that only holds the required information. Based on the information, if you search for it, you will be able to get as part of your hits the encompassing class. Note as well, that when changing the referenced class, for it to be reflected as part of the <code>ref-comp-alias</code> you will have to save all the relevant encompassing classes.
accessor (optional, defaults to property)	The strategy to access the class property value. <code>property</code> access using the Java Bean accessor methods, while <code>field</code> directly access

Attribute	Description
	the class fields.
converter (optional)	The global converter lookup name registered with the configuration.

The reference element defines a "pointer" to a class dependency identified in `ref-alias`.

The type of the JavaBean property can be the class mapping class itself, an `Array` of it, or a `Collection`.

Currently there is no support for lazy behavior or cascading. It means that when saving an object, it will not persist the object defined references and when loading an object, it will load all it's references. Future versions will support lazy and cascading features.

Compass::Core supports cyclic references, which means that two classes can have a cyclic reference defined between them.

### 4.3.10. parent

Declaring and using the `parent` element.

```
<parent
  name="the class property name"
  accessor="property|field"
  converter="converter lookup name"
>
</reference>
```

**Table 4.10.**

Attribute	Description
name	The class property (a.k.a JavaBean property) name, with initial lowercase letter.
accessor (optional, defaults to <code>property</code> )	The strategy to access the class property value. <code>property</code> access using the Java Bean accessor methods, while <code>field</code> directly access the class fields.
converter (optional)	The global converter lookup name registered with the configuration.

The parent mapping provides support for cyclic mappings for components. If the component class mapping wish to map the enclosing class, the parent mapping can be used to map to it. The parent mapping will not marshal (persist the data to the search engine) the parent object, it will only initialize it when loading the parent object from the search engine.

### 4.3.11. constant

Declaring a constant set of meta-data using the `constant` element.

```
<constant
  exclude-from-all="false|true"
  converter="converter lookup name"
```

```
>
  meta-data,
  meta-data-value+
</reference>
```

**Table 4.11.**

Attribute	Description
exclude-from-all (optional, defaults to false)	Excludes the constant meta-data and all its values from participating in the "all" feature.
override (optional, defaults to true)	If there is another definition with the same mapping name, if it will be overridden or added as additional mapping. Mainly used to override definitions made in extended mappings.
converter (optional)	The global converter lookup name registered with the configuration.

If you wish to define a set of constant meta data that will be embedded within the searchable class (`Resource`), you can use the `constant` element. You define the usual `meta-data` element followed by one or more `meta-data-value` elements with the value that maps to the `meta-data` within it.

---

# Chapter 5. OSEM - Annotations

## 5.1. Introduction

Compass::Core provides the ability to map Java Objects to the underlying Search Engine through Java 5 annotations, we call this technology OSEM (Object Search Engine Mapping). OSEM provides a rich syntax for describing Object attributes and relationships. OSEM definitions are used by Compass to extract the required property from the Object model at run-time and inserting the required meta-data into the Search Engine index.

## 5.2. Searchable Classes

Searchable classes are normally classes representing the state of the application, implementing the entities with the business model. Compass works best if the classes follow the simple Plain Old Java Object (POJO) programming model. The following class is an example of a searchable class (with Compass annotations):

```
import java.util.Date;
import java.util.Set;

@Searchable
public class Author {
    @SearchableId
    private Long id; // identifier
    private String name;
    private Date birthday;
    private Set books;

    private void setId(Long id) {
        this.id = id;
    }

    public Long getId() {
        return this.id;
    }

    public void setName(String name) {
        this.name = name;
    }

    @SearchableProperty
    public String getName() {
        return this.name;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    @SearchableProperty(name = "birthdayOrig")
    @SearchableMetaData(name = "birthday" format = "yyyy-MM-dd")
    public Date getBirthday() {
        return this.birthday;
    }

    // @SearchableReference
    @SearchableComponent
    public void setBooks(Set<Book> books) {
        this.books = books;
    }

    public Set<Book> getBooks() {
        return this.books;
    }

    // addBook not needed by Compass::Core
    public void addBook(Book book) {
        this.books.add(book);
    }
}
```

```
}  
}
```

Compass works non-intrusive with application Objects, these Objects must follow several rules:

### 5.2.1. Implement a Default Constructor

`Author` has an implicit default (no-argument) constructor. All persistent classes must have a default constructor (which may be non-public) so `Compass::Core` can instantiate using `Constructor.newInstance()`.

### 5.2.2. Provide Property Identifier(s)

OSEM requires that any mapped Object will define one or more properties (JavaBean properties) that identifies the class. The id properties can be called anything, and it's type can be any primitive type, primitive "wrapper" type, `java.lang.String` or `java.util.Date`. For special cases, the id can be a user defined class, with the class annotated with `@SearchableClassConverter` and a user defined `Converter` (usually extends `CompassAbstractBasicConverter`) responsible for converting to class to a `String`.

### 5.2.3. Declare Accessors and Mutators (Optional)

Even though Compass can directly persist instance variables, it is usually better to decouple this implementation detail from the Search Engine mechanism. `Compass::Core` recognizes JavaBean style property (`getFoo`, `isFoo`, and `setFoo`). This mechanism works with any level of visibility.

### 5.2.4. Implementing equals() and hashCode()

You have to override the `equals()` and `hashCode()` methods if you intend to mix objects of persistent classes (e.g. in a `Set`). You can implement it by using the identifier of both objects, but note that `Compass::Core` works best with surrogate identifier (and will provide a way to automatically generate them), thus it is best to implement the methods using business keys.

## 5.3. Mapping Annotations

The annotations are well documented in Compass javadocs. The following sections will try and explain important aspects of using different annotations. For full documentation, please consult the javadocs.

### 5.3.1. @Searchable

The `@Searchable` annotation marks a class as searchable. It allows to perform full text search on its annotated fields/properties. The searchable class is associated with an alias, which defaults to the shorthand name of the class.

### 5.3.2. @SearchableId

Each `@Searchable` class must have at least one annotated field/property with `@SearchableId`. The type of the `@SearchableId` can be either one of Java primitive types, their corresponding wrappers, or a user defined class. In case of a user defined class, the defined class should have a specialized converter associated with it, with the preferable way of associating a `Converter` is to use the `@SearchableClassConverter` annotating the custom

class (Compass will automatically identify the `Converter`). Also note, the `Converter` will most of times create a `String` representation of the custom class, meaning it is preferable to extend `CompassAbstractBasicConverter`.

It is important to understand how meta-datas are created when using the `@SearchableId` annotation. When the annotation is used without any parameters, no meta-data will be created. As a result, Compass will create its own internal managed meta-data. If the `name` parameter is set, Compass will create a meta-data associated with the name (Compass might still create an additional internal meta-data). The annotation allows for all the aspects of the meta-data to be controlled using the `@SearchableId` annotation (the parameters are clearly defined in the javadoc, and match the `@SearchableMetaData` parameters). For more than one meta-data, use the `@SearchableMetaData` and `@SearchableMetaDatas` annotations on top of the `@SearchableId` annotation.

### 5.3.3. @SearchableProperty

Defines a searchable property on a `@Searchable` class field/property. The `@SearchableProperty` is meant to handle basic types (which usually translate to a `String` saved in the search engine index).

Again, it is important to understand how meta-datas are created when using the `@SearchableProperty` annotation. If no parameters are set, the searchable property will automatically create a searchable meta-data (not Compass internal meta-data, which might be created as well), with its name being the class field/property name. It will NOT create a meta-data automatically if the `name` is NOT set AND there are either `@SearchableMetaData` or `@SearchableMetaDatas` annotating the class field/property. You can control the auto-generated meta-data using the searchable property parameters (clearly marked in the javadoc), and they map one to one with the `@SearchableMetaData` parameters.

The searchable property can annotate a `Java Collection` type field/property (with its element a basic type), supporting `Java List` and `Set`. Compass will try and auto identify the type if using Generics, and if the collection does not use generics, the `type` parameter should be used to explicitly define the collection element type. It can also annotation an `Array` type field/property (with its element a basic type).

### 5.3.4. @SearchableComponent

A searchable component is a class field/property that reference another class, which content need to be embedded into the content of its `Searchable` class. It will results in searches performed on the component class to return the component field/property searchable class. The referenced class must have searchable definitions, defined either using annotations or other means (like xml).

The searchable component can annotate a `Java Collection` type field/property, supporting either `List` or `Set`. The searchable component will try and automatically identify the element type using generics, but if the collection is not defined with generics, `refAlias` parameter should be used to reference the component searchable class mapping definitions. The searchable component can annotate an array as well, with the array element type used for referenced searchable class mapping definitions.

### 5.3.5. @SearchableReference

A searchable reference is a class field/property that reference another class, and the relationship need to be stored by Compass so it can be traversed when getting the class from the index. Compass will end up saving only the ids of the referenced class in the search engine index.

The searchable reference can annotate a `Java Collection` type field/property, supporting either `List` or `Set`. The searchable reference will try and automatically identify the element type using generics, but if the collection is not defined with generics, `refAlias` parameter should be used to reference the referenced

searchable class mapping definitions. The searchable reference can annotate an array as well, with the array element type used for referenced searchable class mapping definitions.

## 5.4. Configuration Annotations

Compass also allows using annotation for certain configuration settings. The annotations are defined on a package level (`package-info.java`). Some of the configuration annotations are `@SearchAnalyzer`, `@SearchAnalyzerFilter`, and `@SearchConverter`. Please see the javadocs for more information.

---

# Chapter 6. XSEM - Xml to Search Engine Mapping

## 6.1. Introduction

Compass::Core provides the ability to map XML structure to the underlying Search Engine through simple XML mapping files, we call this technology XSEM (XML to Search Engine Mapping). XSEM provides a rich syntax for describing XML mappings using Xpath expressions. The XSEM files are used by Compass to extract the required xml elements from the xml structure at run-time and inserting the required meta-data into the Search Engine index.

## 6.2. Xml Object

At the core of XSEM supports is `XmlObject` abstraction on top of the actual XML library implementation. The `XmlObject` represents an XML element (document, node, attribute, ...) which is usually the result of an Xpath expression. It allows to get the name and value of the given element, and execute Xpath expressions against it (for more information please see the `XmlObject` javadoc).

Here is an example of how `XmlObject` is used with Compass:

```
CompassSession session = compass.openSession();
// ...
XmlObject xmlObject = // create the actual XmlObject implementation (we will see how soon)
session.save("alias", xmlObject);
```

An extension to the `XmlObject` interface is the `AliasedXmlObject` interface. It represents an xml object that is also associated with an alias. This means, that saving the object does not require to explicitly specify the alias that it will be saved under.

```
CompassSession session = compass.openSession();
// ...
AliasedXmlObject xmlObject = // create the actual XmlObject implementation (we will see how soon)
session.save(xmlObject);
```

Compass comes with support for dom4j and JSE 5 xml libraries, here is an example of how to use dom4j API in order to create a dom4j xml object:

```
CompassSession session = compass.openSession();
// ...
SAXReader saxReader = new SAXReader();
Document doc = saxReader.read(new StringReader(xml));
AliasedXmlObject xmlObject = new Dom4jAliasedXmlObject(alias, doc.getRootElement());
session.save(xmlObject);
```

And here is a simple example of how to use JSE 5:

```
CompassSession session = compass.openSession();
// ...
Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(new InputSource(new StringReader(xml)));
AliasedXmlObject xmlObject = NodeAliasedXmlObject(alias, doc);
session.save(xmlObject);
```

## 6.3. Xml Content Handling

Up until now, Compass has no knowledge of how to parse and create an actual `XmlObject` implementation, or how to convert an `XmlObject` into its xml representation. This is perfectly fine, but it also means that systems will not be able to work with `XmlObject` for read/search operations. Again, this is perfectly ok for some application, since they can always work with the underlying `Resource` representation, but some applications would still like to store the actual xml content in the search engine, and work with the `XmlObject` for read/search operations.

Compass XSEM support allows to define the `xml-content` mapping (defined below), which will cause Compass to store the xml representation in the search engine as well. It will also mean that for read/search operations, the application will be able to get an `XmlObject` back (for example, using `CompassSession#get` operation).

In order to support this, Compass must be configured with how to parse the xml content into an `XmlObject`, and how to convert an `XmlObject` into an xml string. Compass comes with built in converters that do exactly that:

**Table 6.1. Compass XmlContentConverters**

XmlContentConverter	Description
<code>org.compass.core.xml.javax.converter.NoSupportForJSE5Converter</code>	Support for JSE 5 xml libraries. Not recommended on account of performance.
<code>org.compass.core.xml.dom4j.converter.SAXReaderXmlContentConverter</code>	Supports <code>SAXReader</code> for parsing, and <code>XMLWriter</code> to write the raw xml data.
<code>org.compass.core.xml.dom4j.converter.XMLReaderXmlContentConverter</code>	Supports <code>XMLReader</code> for parsing, and <code>XMLWriter</code> to write the raw xml data.
<code>org.compass.core.xml.dom4j.converter.XMLPullParserXmlContentConverter</code>	Supports <code>XMLPullParser</code> for parsing, and <code>XMLWriter</code> to write the raw xml data.

Most of the time, better performance can be achieved by pooling `XmlContentConverters` implementations. Compass handling of `XmlContentConverter` allows for three different instantiation models: prototype, pool, and singleton. prototype will create a new `XmlContentConverter` each time, a singleton will use a shared `XmlContentConverter` for all operations, and pooled will pool `XmlContentConverter` instances. The default is prototype.

Here is an example of a Compass schema based configuration that registers a global Xml Content converter:

```
<compass-core-config ...
  <compass name="default">

    <connection>
      <file path="target/test-index" />
    </connection>

    <converters>
      <converter name="compass.converter.xmlContentMapping"
        type="org.compass.core.converter.mapping.xsem.XmlContentMappingConverter">
        <setting name="xmlContentConverter.type" value="[fully qualified class name of XmlContentConverter]" />
        <setting name="xmlContentConverter.wrapper" value="prototype" />
      </converter>
    </converters>

  </compass>
</compass-core-config>
```

And here is an example of a DTD (settings) based configuration file:

```

<!DOCTYPE compass-core-configuration PUBLIC ...
<compass-core-configuration>
  <compass>
    <setting name="compass.converter.xmlContentMapping.type">
      org.compass.core.converter.mapping.xsem.XmlContentMappingConverter
    </setting>
    <setting name="compass.converter.xmlContentMapping.xmlContentConverter.type">
      [fully qualified class name of XmlContentConverter]
    </setting>
    <setting name="compass.converter.xmlContentMapping.xmlContentConverter.wrapper">
      prototype
    </setting>
  </compass>
</compass-core-configuration>

```

And last, here is how it can be configured it programmatically:

```

settings.setGroupSettings(CompassEnvironment.Converter.PREFIX,
    CompassEnvironment.Converter.DefaultTypeNames.Mapping.XML_CONTENT_MAPPING,
    new String[]{CompassEnvironment.Converter.TYPE, CompassEnvironment.Converter.XmlContent.TYPE},
    new String[]{XmlContentMappingConverter.class.getName(), XPP3ReaderXmlContentConverter.class.getName()});

```

Note, that specific converters can be associated with a specific `xml-object` mapping, in order to do it, simply register the converter under a different name (`compass.converter.xmlContentMapping` is the default name that Compass will use when nothing is configured), and use that name in the converter attribute of the `xml-content` mapping.

## 6.4. Raw Xml Object

If Compass is configured with an Xml Content converter, it now knows how to parse an xml content into an `XmlObject`. This allows us to simplify more the creation of `XmlObjects` from a raw xml data. Compass comes with a wrapper `XmlObject` implementation, which handles raw xml data (non parsed one). Here is how it can be used:

```

Reader xmlData = // construct an xml reader over raw xml content
AliasedXmlObject xmlObject = RawAliasedXmlObject(alias, xmlData);
session.save(xmlObject);

```

Here, Compass will identify that it is a `RawAliasedXmlObject`, and will use the registered converter (or the one configured against the `xml-content` mapping for the given alias) to convert it to the appropriate `XmlObject` implementation. Note, that when performing any read/search operation, the actual `XmlObject` that will be returned is the one the the registered converter creates, and not the raw xml object.

## 6.5. Mapping Definition

XML/Search Engine mappings are defined in an XML document, and maps XML data structures. The mappings are xml centric, meaning that mappings are constructed around XML data structures themselves and not internal Resources. If we take the following as a sample XML data structure:

```

<xml-fragment>
  <data>
    <id value="1"/>
    <data1 value="data1attr">data1</data1>
    <data1 value="data12attr">data12</data1>
  </data>
  <data>
    <id value="2"/>
    <data1 value="data21attr">data21</data1>
    <data1 value="data22attr">data22</data1>
  </data>
</xml-fragment>

```

```
</data>
</xml-fragment>
```

We can map it using the following XSEM definition file:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
  "-//Compass/Compass Core Mapping DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-mapping.dtd">

<compass-core-mapping>

  <xml-object alias="data1" xpath="/xml-fragment/data[1]">
    <xml-id name="id" xpath="id/@value" />
    <xml-property xpath="data1/@value" />
    <xml-property name="eleText" xpath="data1" />
  </xml-object>

  <xml-object alias="data2" xpath="/xml-fragment/data">
    <xml-id name="id" xpath="id/@value" />
    <xml-property xpath="data1/@value" />
    <xml-property name="eleText" xpath="data1" />
  </xml-object>

  <xml-object alias="data3" xpath="/xml-fragment/data">
    <xml-id name="id" xpath="id/@value" />
    <xml-property xpath="data1/@value" />
    <xml-property name="eleText" xpath="data1" />
    <xml-content name="content" />
  </xml-object>

</compass-core-mapping>
```

The mapping definition here shows three different mappings (that will work with the sample xml). The different mappings are registered under different aliases, where the alias acts as the connection between the actual XML saved and the mappings definition.

An `xml-object` mapping can have an associated `xpath` expression with it, which will narrow down the actual xml elements that will represent the top level xml object which will be mapped to the search engine. A nice benefit here, is that the `xpath` can return multiple xml objects, which in turn will result in multiple `Resources` saved to the search engine.

Each xml object mapping must have at least one `xml-id` mapping definition associated with it. It is used in order to update/delete existing xml objects.

In the mapping definition associated with `data3` alias, the `xml-content` mapping is used, which stores the actual xml content in the search engine as well. This will allow to unmarshall the xml back into an `XmlObject` representation. For the first two mappings (`data1` and `data2`), search/read operations will only be able to work on the `Resource` level.

### 6.5.1. xml-object

You may declare a xml object mapping using the `xml-object` element:

```
<xml-object
  alias="aliasName"
  sub-index="sub index name"
  xpath="optional xpath expression"
  analyzer="name of the analyzer"
  all="true|false"
  all-term-vector="no|yes|positions|offsets|positios_offsets"
  all-metadata="name of all metadata"
/>
xml-id*,
```

```
(xml-analyzer?),
(xml-property)*,
(xml-content?)
```

**Table 6.2. xml-object mapping**

Attribute	Description
alias	The name of the alias that represents the <code>xmlObject</code> .
sub-index (optional, defaults to the alias value)	The name of the sub-index that the alias will map to.
xpath (optional, will not execute an xpath expression if not specified)	An optional xpath expression to narrow down the actual xml elements that will represent the top level xml object which will be mapped to the search engine. A nice benefit here, is that the xpath can return multiple xml objects, which in turn will result in multiple <code>Resources</code> saved to the search engine.
analyzer (optional, defaults to the default analyzer)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> properties. Defaults to the <code>default</code> analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the <code>xml-analyzer</code> mapping (a child mapping of xml object mapping) (for an xml element that controls the analyzer), the analyzer attribute will have no effects.
all (optional, defaults to <code>true</code> )	Specifies if when persisting the <code>xmlObject</code> , <code>Compass::Core</code> will create the all property (which represents all the properties/meta-data values).
all-term-vector (optional, defaults to configuration setting <code>compass.property.all.termVector</code> )	The term vector value of the all property.
all-metadata (optional, defaults to the setting)	The name of the all meta-data (property).

## 6.5.2. xml-id

Mapped `xmlObject`'s must declare at least one `xml-id`. The `xml-id` element defines the `xmlObject` (element, attribute, ...) that identifies the root `xmlObject` for the specified alias.

```
<xml-id
  name="the name of the xml id"
  xpath="xpath expression"
  value-converter="value converter lookup name"
  converter="converter lookup name"
/>
```

**Table 6.3. xml-id mapping**

Attribute	Description
name	The name of the <code>xml-id</code> . Will be used when constructing the <code>xml-id</code>

Attribute	Description
	internal path.
xpath	The xpath expression used to identify the xml-id. Must return a single xml element.
value-converter (optional, default to Compass SimpleXmlValueConverter)	The global converter lookup name registered with the configuration. This is a converter associated with converting the actual value of the xml-id. Acts as a convenient extension point for custom value converter implementation (for example, date formatters). SimpleXmlValueConverter will usually act as a base class for such extensions.
converter (optional)	The global converter lookup name registered with the configuration. The converter will is responsible to convert the xml-id mapping.

An important note regarding the `xml-id` mapping, is that it will always at as an internal Compass `Property`. This means that if one wish to have it as part of the searchable content, it will have to be mapped with `xml-property` as well.

### 6.5.3. xml-property

Declaring and using the `xml-property` element.

```
<xml-property
  xpath="xpath expression"
  name="optionally the name of the xml property"
  store="yes|no|compress"
  index="tokenized|un_tokenized|no"
  boost="boost value for the property"
  analyzer="name of the analyzer"
  reverse="no|reader|string"
  override="true|false"
  exclude-from-all="false|true"
  value-converter="value converter lookup name"
  converter="converter lookup name"
/>
```

**Table 6.4. xml-property mapping**

Attribute	Description
name (optional, will use the xml object (element, attribute, ...) name if not set)	The name that the value will be saved under. It is optional, and if not set, will use the xml object name (the result of the xpath expression).
xpath	The xpath expression used to identify the xml-property. Can return no xml objects, one xml object, or many xml objects.
store (optional, defaults to <code>yes</code> )	If the value of the xml property is going to be stored in the index.
index (optional, defaults to <code>tokenized</code> )	If the value of the xml property is going to be indexed (searchable). If it does, than controls if the value is going to be broken down and analyzed ( <code>tokenized</code> ), or is going to be used as is ( <code>un_tokenized</code> ).
boost (optional, defaults to <code>1.0f</code> )	Controls the boost level for the xml property.

Attribute	Description
analyzer (optional, defaults to the xml mapping analyzer decision scheme)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> xml property mappings defined for the given property. Defaults to the xml mapping analyzer decision scheme based on the analyzer set, or the <code>xml-analyzer</code> mapping.
exclude-from-all (optional, default to false)	Excludes the property from participating in the "all" meta-data.
override (optional, defaults to true)	If there is another definition with the same mapping name, if it will be overridden or added as additional mapping. Mainly used to override definitions made in extended mappings.
reverse (optional, defaults to no)	The meta-data will have it's value reversed. Can have the values of <code>no</code> - no reverse will happen, <code>string</code> - the reverse will happen and the value stored will be a reversed string, and <code>reader</code> - a special reader will wrap the string and reverse it. The <code>reader</code> option is more performant, but the <code>store</code> and <code>index</code> settings will be discarded.
value-converter (optional, default to Compass SimpleXmlValueConverter)	The global converter lookup name registered with the configuration. This is a converter associated with converting the actual value of the xml-id. Acts as a convenient extension point for custom value converter implementation (for example, date formatters). <code>SimpleXmlValueConverter</code> will usually act as a base class for such extensions.
converter (optional)	The global converter lookup name registered with the configuration. The converter will is responsible to convert the xml-property mapping.

#### 6.5.4. xml-analyzer

Declaring an analyzer controller property using the `xml-analyzer` element.

```
<xml-analyzer
  name="property name"
  xpath="xpath expression"
  null-analyzer="analyzer name if value is null"
  converter="converter lookup name"
>
</xml-analyzer>
```

**Table 6.5. xml-analyzer mapping**

Attribute	Description
name	The name of the xml-analyzer (results in a <code>Property</code> ).
xpath	The xpath expression used to identify the xml-analyzer. Must return a single xml element.
null-analyzer (optional, defaults to error in case of a null value)	The name of the analyzer that will be used if the property has a null value, or the xpath expression returned no elements.
converter (optional)	The global converter lookup name registered with the

Attribute	Description
	configuration.

The analyzer xml property mapping, controls the analyzer that will be used when indexing the `xmlObject`. If the mapping is defined, it will override the xml object mapping analyzer attribute setting.

If, for example, Compass is configured to have two additional analyzers, called `an1` (and have settings in the form of `compass.engine.analyzer.an1.*`), and another called `an2`. The values that the xml property can hold are: `default` (which is an internal Compass analyzer, that can be configured as well), `an1` and `an2`. If the analyzer will have a `null` value, and it is applicable with the application, a `null-analyzer` can be configured that will be used in that case. If the resource property has a value, but there is not matching analyzer, an exception will be thrown.

### 6.5.5. xml-content

Declaring an xml content mapping using the `xml-content` element.

```
<xml-content
  name="property name"
  store="yes|compress"
  converter="converter lookup name"
>
</xml-content>
```

**Table 6.6. xml-content mapping**

Attribute	Description
name	The name the xml content will be saved under.
store (optional, defaults to yes)	How to store the actual xml content.
converter (optional)	The global converter lookup name registered with the configuration.

The `xml-content` mapping causes Compass to store the actual xml content in the search engine as well. This will allow to unmarshal the xml back into an `xmlObject` representation. For `xml-object` mapping without an `xml-content` mapping, search/read operations will only be able to work on the `Resource` level.

---

# Chapter 7. Resource Mapping

## 7.1. Introduction

Compass::Core provides OSEM technology for use with an applications Object domain model. Compass::Core also provides Resource Mapping technology for resources other than Objects (that do not benefit from OSEM). The benefits of using Resources can be summarized as:

- Your application does not have a domain model (therefore cannot use OSEM), but you still want to use the functionality of Compass.
- Your application already works with Lucene, but you want to add Compass::Core additional features (i.e. transactions, fast updates). Working with Resources makes your migration easy.
- You execute a query and want to update all the meta-data (Resource Property) with a certain value. You use OSEM in your application, but you do not wish to iterate through the results, performing run-time object type checking and casting to the appropriate object type before method call. You can simply use the Resource interface and treat all the results in the same abstracted way.

## 7.2. Mapping Declaration

In order to work directly with a Resource, Compass needs to know the alias and the primary properties (i.e. primary keys in data-base systems) associated with the Resource. The primary properties are also known as id properties. This information is declared in Resource Mapping XML documents, so that Compass knows how to manage the Resource internally.

Here is an example of a Resource Mapping XML document:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
  "-//Compass/Compass Core Mapping DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-mapping.dtd">

<compass-core-mapping>

  <resource alias="a">
    <resource-id name="id" />
  </resource>

  <resource alias="b">
    <resource-id name="id1" />
    <resource-id name="id2" />
  </resource>

  <resource alias="c">
    <resource-id name="id" />
    <resource-property name="value1" />
    <resource-property name="value2" store="yes" index="tokenized" />
    <resource-property name="value3" store="compress" index="tokenized" />
    <resource-property name="value4" store="yes" index="un_tokenized" />
    <resource-property name="value5" store="yes" index="no" />
  </resource>
</compass-core-mapping>
```

Now that the Resource Mapping has been declared, you can create the Resource in the application. In the following code example the Resource is created with an alias and id property matching the Resource Mapping declaration.

```

Resource r = session.createResource("a");
Property id = session.createProperty("id", "1",
    Property.Store.YES, Property.Index.UN_TOKENIZED);
r.addProperty(id);
r.addProperty(session.createProperty("mvalue", "property test",
    Property.Store.YES, Property.Index.TOKENIZED));

session.save(r);

```

The Resource Mapping file example above defines mappings for two resources (each identified with a different alias). Each resource has a set of resource ids that are associated with it. The value for the `resource-id` tag is the name of the `Property` that is associated with the primary property for the `Resource`.

The third mapping (alias "c"), defines `resource-property` mappings as well as `resource-id` mappings. The `resource-property` mapping works with the `Resource#addProperty(String name, Object value)` operation. It provides definitions for the resource properties that are added (index, store, and so on), and they are then looked up when using the mentioned add method. Using the `resource-property` mapping, helps clean up the code when constructing a `Resource`, since all the `Property` characteristics are defined in the mapping definition, as well as auto conversion from different objects, and the ability to define new ones. Note that the `resource-property` definition will only work with the mentioned `addProperty` method, and no other `addProperty` method.

All XML mappings should declare the doctype shown. The actual DTD may be found at the URL above or in the compass core distribution. Compass will always look for the DTD in the classpath first.

There are no `compass-core-mapping` attributes that are applicable when working with resource mappings.

## 7.2.1. resource

You may declare a resource mapping using the `resource` element:

```

<resource
  alias="aliasName"
  sub-index="sub index name"
  extends="a comma separated list of aliases to extend"
  analyzer="name of the analyzer"
  all="true|false"
  all-term-vector="no|yes|positions|offsets|positios_offsets"
  all-metadata="name of all metadata"
/>
resource-id*,
(resource-analyzer?),
(resource-property)*

```

**Table 7.1.**

Attribute	Description
alias	The name of the alias that represents the <code>Resource</code> .
sub-index (optional, defaults to the alias value)	The name of the sub-index that the alias will map to.
extends (optional)	A comma separated list of aliases to extend. Can extend a <code>resource</code> mapping or a <code>resource-contract</code> mapping. Note that can extend more than one <code>resource/resource-contract</code>
analyzer (optional, defaults to the	The name of the analyzer that will be used to analyze <code>TOKENIZED</code>

Attribute	Description
default analyzer)	properties. Defaults to the <code>default</code> analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the <code>resource-analyzer</code> mapping (a child mapping of resource mapping) (for a resource property value that controls the analyzer), the analyzer attribute will have no effects.
all (optional, defaults to <code>true</code> )	Specifies if when persisting the <code>Resource</code> , <code>Compass::Core</code> will create the all property (which represents all the properties/meta-data values).
all-term-vector (optional, defaults to configuration setting <code>compass.property.all.termVector</code> )	The term vector value of the all property.
all-metadata (optional, defaults to the setting)	The name of the all meta-data (property).

## 7.2.2. resource-contract

You may declare a resource mapping contract using the `resource-contract` element:

```
<resource-contract
  alias="aliasName"
  extends="a comma separated list of aliases to extend"
  analyzer="name of the analyzer"
/>
resource-id*,
(resource-analyzer?),
(resource-property)*
```

**Table 7.2.**

Attribute	Description
alias	The name of the alias that represents the <code>Resource</code> .
extends (optional)	A comma separated list of aliases to extend. Can extend a <code>resource</code> mapping or a <code>resource-contract</code> mapping. Note that can extend more than one <code>resource/resource-contract</code>
analyzer (optional, defaults to the default analyzer)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> properties. Defaults to the <code>default</code> analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the <code>resource-analyzer</code> mapping (a child mapping of resource mapping) (for a resource property value that controls the analyzer), the analyzer attribute will have no effects.

## 7.2.3. resource-id

Mapped `Resource`'s must declare at least one `resource-id`. The `resource-id` element defines the `Property` that identifies the `Resource` for the specified alias.

```
<resource-id
  name="idName"
/>
```

Table 7.3.

Attribute	Description
name	The name of the <code>Property</code> (known also as the name of the meta-data) that is the id of the <code>Resource</code> .

## 7.2.4. resource-property

Declaring and using the `resource-property` element.

```
<resource-property
  name="property name"
  store="yes|no|compress"
  index="tokenized|un_tokenized|no"
  boost="boost value for the property"
  analyzer="name of the analyzer"
  reverse="no|reader|string"
  override="true|false"
  exclude-from-all="[parent's exclude-from-all]|false|true"
  converter="converter lookup name"
>
</resource-property>
```

Table 7.4.

Attribute	Description
name	The name of the <code>Property</code> (known also as the name of the meta-data).
store (optional, defaults to <code>yes</code> )	If the value of the resource property is going to be stored in the index.
index (optional, defaults to <code>tokenized</code> )	If the value of the resource property is going to be indexed (searchable). If it does, than controls if the value is going to be broken down and analyzed ( <code>tokenized</code> ), or is going to be used as is ( <code>un_tokenized</code> ).
boost (optional, defaults to <code>1.0f</code> )	Controls the boost level for the resource property.
analyzer (optional, defaults to the resource mapping analyzer decision scheme)	The name of the analyzer that will be used to analyze <code>TOKENIZED</code> resource property mappings defined for the given property. Defaults to the resource mapping analyzer decision scheme based on the analyzer set, or the <code>resource-analyzer</code> mapping.
exclude-from-all (optional, default to <code>false</code> )	Excludes the property from participating in the "all" meta-data.
override (optional, defaults to <code>true</code> )	If there is another definition with the same mapping name, if it will be overridden or added as additional mapping. Mainly used to override definitions made in extended mappings.

Attribute	Description
reverse (optional, defaults to no)	The meta-data will have it's value reversed. Can have the values of <code>no</code> - no reverse will happen, <code>string</code> - the reverse will happen and the value stored will be a reversed string, and <code>reader</code> - a special reader will wrap the string and reverse it. The <code>reader</code> option is more perform ant, but the <code>store</code> and <code>index</code> settings will be discarded.
converter (optional)	The global converter lookup name registered with the configuration.

Defines the characteristics of a `Resource Property` identified by the `name` mapping. The definition only applies when using the `Resource#addProperty(String name, Object value)` operation, and the operation can only be used with the `resource-property` mapping.

Note that other `Resource Property` can be added that are not defined in the resource mapping using the `createProperty` operation.

## 7.2.5. resource-analyzer

Declaring an analyzer controller property using the `resource-analyzer` element.

```
<resource-analyzer
  name="property name"
  null-analyzer="analyzer name if value is null"
  converter="converter lookup name"
>
</resource-analyzer>
```

**Table 7.5.**

Attribute	Description
name	The name of the <code>Property</code> (known also as the name of the meta-data).
null-analyzer (optional, defaults to error in case of a null value)	The name of the analyzer that will be used if the property has the null value.
converter (optional)	The global converter lookup name registered with the configuration.

The analyzer resource property mapping, controls the analyzer that will be used when indexing the `Resource`. If the mapping is defined, it will override the resource mapping analyzer attribute setting.

If, for example, Compass is configured to have two additional analyzers, called `an1` (and have settings in the form of `compass.engine.analyzer.an1.*`), and another called `an2`. The values that the resource property can hold are: `default` (which is an internal Compass analyzer, that can be configured as well), `an1` and `an2`. If the analyzer will have a null value, and it is applicable with the application, a `null-analyzer` can be configured that will be used in that case. If the resource property has a value, but there is not matching analyzer, an exception will be thrown.

---

# Chapter 8. Common Meta Data

## 8.1. Introduction

The common meta-data feature of Compass::Core provides a way to externalize the definition of meta-data names and aliases used in OSEM files, especially useful if your application has a large domain model with many OSEM files. Another advantage of this mechanism is the ability to add extra information to the meta data (i.e. a long description) and the ability to specify the format for the meta-data definition, removing the need to explicitly define formats in the OSEM file (like `...format="yyyy/MM/dd" . .`).

By centralizing your meta-data, other tools can take advantage of this information and extend this knowledge (i.e. adding semantic meaning to the data). Compass::Core provides a common meta-data Ant task that generates a Java class containing constant values of the information described in the Common meta-data file, allowing programmatic access to this information from within the application (see Library class in sample application).

Note, the common meta-data support in Compass is completely optional for applications.

## 8.2. Common Meta Data Definition

The common meta-data definition are defined in an XML document. Here is an example:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-meta-data PUBLIC
  "-//Compass/Compass Core Meta Data DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-meta-data.dtd">

<compass-core-meta-data>

  <meta-data-group id="library" displayName="Library Meta Data">

    <description>Library Meta Data</description>
    <uri>http://compass/sample/library</uri>

    <alias id="author" displayName="Author">
      <description>Author alias</description>
      <uri>http://compass/sample/library/alias/author</uri>
      <name>author</name>
    </alias>

    <alias id="name" displayName="Name">
      <description>Name alias</description>
      <uri>http://compass/sample/library/alias/name</uri>
      <name>name</name>
    </alias>

    <alias id="article" displayName="Article">
      <description>Article alias</description>
      <uri>http://compass/sample/library/alias/article</uri>
      <name>article</name>
    </alias>

    <alias id="book" displayName="Book">
      <description>Book alias</description>
      <uri>http://compass/sample/library/alias/book</uri>
      <name>book</name>
    </alias>

    <meta-data id="type" displayName="Type">
      <description>Type of an entity in the system</description>
      <uri>http://compass/sample/library/type</uri>
      <name>type</name>
      <value id="mdPerson">person</value>
      <value id="mdAuthor">author</value>
    </meta-data>
  </meta-data-group>
</compass-core-meta-data>
```

```

</meta-data>

<meta-data id="keyword" displayName="Keyword">
  <description>Keyword associated with an entity</description>
  <uri>http://compass/sample/library/keyword</uri>
  <name>keyword</name>
</meta-data>

<meta-data id="name" displayName="Name">
  <description>The name of a person</description>
  <uri>http://compass/sample/library/name</uri>
  <name>name</name>
</meta-data>

<meta-data id="birthdate" displayName="Birthdate">
  <description>The birthdate of a person</description>
  <uri>http://compass/sample/library/birthdate</uri>
  <name format="yyyy/MM/dd">birthdate</name>
</meta-data>

<meta-data id="isbn" displayName="ISBN">
  <description>ISBN of the book</description>
  <uri>http://compass/sample/library/isbn</uri>
  <name>isbn</name>
</meta-data>

<meta-data id="title" displayName="Title">
  <description>The title of a book or an article</description>
  <uri>http://compass/sample/library/title</uri>
  <name>title</name>
</meta-data>

...

</meta-data-group>

</compass-core-meta-data>

```

### 8.3. Using the Definition

In order to use the Common meta-data definition, you need to specify the location of the file or files in the Compass configuration file (compass.cfg.xml). Compass will automatically replace labels used in OSEM files with the values contain in the Common meta-data file.

```

<meta-data resource=
  "org/compass/sample/library/library.cmd.xml" />

```

Note: The common meta data reference needs to be BEFORE the mapping files that use them.

To use common meta data within a OSEM file, you use the familiar `${...}` label (similar to Ant). An example of using the common meta data definitions in the mapping file is:

```

<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
  "-//Compass/Compass Core Mapping DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-mapping.dtd">
<compass-core-mapping package="org.compass.sample.library">

  <class name="Author" alias="${library.author}">

    <id name="id" />

    <constant>
      <meta-data>${library.type}</meta-data>
      <meta-data-value>${library.type.mdPerson}</meta-data-value>
      <meta-data-value>${library.type.mdAuthor}</meta-data-value>
    </constant>

```

```

<property name="keywords">
  <meta-data boost="2">${library.keyword}</meta-data>
</property>

<component name="name" ref-alias="${library.name}" />

<property name="birthdate">
  <meta-data>${library.birthdate}</meta-data>
</property>

<component name="articles" ref-alias="${library.article}" />

<reference name="books" ref-alias="${library.book}" />

</class>

<class name="Name" alias="${library.name}" root="false">
<property name="title">
  <meta-data>${library.titleName}</meta-data>
</property>
<property name="firstName">
  <meta-data>${library.firstName}</meta-data>
  <meta-data>${library.name}</meta-data>
</property>
<property name="lastName">
  <meta-data>${library.lastName}</meta-data>
  <meta-data>${library.name}</meta-data>
</property>
</class>

</compass-core-mapping>

```

## 8.4. Common Meta Data Ant Task

One of the benefits of using the common meta data definitions is the meta data Ant task, which generate Java classes with constant values of the defined definitions. The common meta data classes allows you to use the definition within your code.

The following is a snippet from an ant build script (or maven) which uses the common meta data ant task.

```

<taskdef name="mdtask"
  classname="org.compass.core.metadata.ant.MetadataTask"
  classpathref="classpathref"/>
<mdtask destdir="${java.src.dir}">
  <fileset dir="${java.src.dir}">
    <include name="**/*" />
  </fileset>
</mdtask>

```

---

# Chapter 9. Transaction

## 9.1. Introduction

As we explained in the overview page, `Compass::Core` provides an abstraction layer on top of the actual transaction handling using the `CompassTransaction` interface. `Compass::Core` has a transaction handling framework in place to support different transaction strategies and comes built in with `LocalTransaction` and `JTA` synchronization support.

As oppose to transaction handling based on `JDBC` data source or `JCA` based resources (and until compass will implement something similar to `JCA`), you have to use the `CompassTransaction` abstraction. Note, that it is made much simpler when using `CompassTemplate` and `CompassCallback` classes since both the session management and the transaction management is done by the template class.

## 9.2. Session Lifecycle

`Compass::Core` `Compass` interface manages the creation of `CompassSession` using the `openSession()` method. When `beginTransaction()` is called on the `CompassTransaction`, the session is bound to the created transaction (`JTA` or `Local`) and used throughout the life-cycle of the transaction. It means that if an additional session is opened within the current transaction, the originating session will be returned by the `openSession()` method.

## 9.3. Local Transaction

`Compass::Core` provides support for compass local transactions. Local transactions are `Compass` session level transaction, with no knowledge of other running transactions (like `JDBC` or `JTA`).

A local transaction which starts within the boundaries of a compass local transaction will share the same session and transaction context and will be controlled by the outer transaction.

In order to configure `Compass` to work with the `Local Transaction`, you must set the `compass.transaction.factory` to `org.compass.core.transaction.LocalTransactionFactory`.

## 9.4. JTA Synchronization Transaction

`Compass::Core` provides support for `JTA` transactions, using the `JTA` synchronization support. A `JTA` transaction will be joined if already started (by `CMT` for example) or will be started if non was initiated.

The support for `JTA` also includes support for suspend and resume provided by the `JTA` transaction manager (or `REQUIRES_NEW` in `CMT` when there is already a transaction running).

`JTA` transaction support is best used when wishing to join with other transactional resources (like `DataSource`).

The current implementation performs the full transaction commit (first and second phase) at the `afterCompletion` method and any exception is logged but not propagated.

In order to configure `Compass` to work with the `JTA Sync Transaction`, you must set the `compass.transaction.factory` to `org.compass.core.transaction.JTASyncTransactionFactory`. You must

also set the transaction manager lookup based on the environment your application will be running at.

If you wish to unit test your application without a container, you might consider using [JOTM](#) as your JTA transaction manager.

---

# Chapter 10. Working with objects

## 10.1. Introduction

Lets assume you have download and configured Compass within your application and create some OSEM mappings. This section provides the basics of how you will use Compass from within the application to load, search and delete Compass searchable objects. All operations within `Compass::Core` are accessed through the `CompassSession` interface. The interface provides `Object` and `Resource` method API's, giving the developer the choice to work directly with `Compass::Core` internal representation (`Resource`) or application domain Objects.

## 10.2. Making Object/Resource Searchable

Newly instantiated objects (or Resources) are saved to the index using the `save(Object)` method. If you have created more than one mapping (alias) to the same object (in OSEM file), use the `save(String alias, Object)` instead.

```
Author author = new Author();
author.setId(new Long(1));
author.setName("Jack London");
compassSession.save(author);
```

## 10.3. Loading an Object/Resource

The `load()` method allows you to load an object (or a Resource) if you already know it's identifier. If you have one mapping for the object (hence one alias), you can use the `load(Class, Object id)` method. If you created more than one mapping (alias) to the same object, use the `load(String alias, Object id)` method instead.

```
Author author = (Author) session.load(Author.class,
    new Long(12));
```

`load()` will throw an exception if no object exists in the index. If you are not sure that there is an object that maps to the supplied id, use the `get` method instead.

## 10.4. Deleting an Object/Resource

If you wish to delete an object (or a Resource), you can use the `delete()` method on the `CompassSession` interface (note that only the identifiers need to be set on the corresponding object or Resource).

## 10.5. Searching

For a quick way to query the index, use the `find()` method. The `find()` method returns a `CompassHits` object, which is an interface which encapsulates the search results. For more control over how the query will executed, use the `CompassQuery` interface, explained later in the section.

```
CompassHits hits = session.find("name:jack");
```

### 10.5.1. Query String Syntax

The free text query string has a specific syntax. The syntax is the same one [Lucene](#) uses, and is summarised here:

**Table 10.1.**

Expression	Hits That
jack	Contain the term <code>jack</code> in the default search field
jack london (jack or london)	Contains the term <code>jack</code> or <code>london</code> , or both, in the default search field
+jack +london (jack and london)	Contains both <code>jack</code> and <code>london</code> in the default search field
name:jack	Contains the term <code>jack</code> in the <code>name</code> property (meta-data)
name:jack -city:london (name:jack and not city:london)	Have <code>jack</code> in the <code>name</code> property and don't have <code>london</code> in the <code>city</code> property
name:"jack london"	Contains the exact phrase <code>jack london</code> in the <code>name</code> property
name:"jack london"~5	Contain the term <code>jack</code> and <code>london</code> within five positions of one another
jack*	Contain terms that begin with <code>jack</code>
jack~	Contains terms that are close to the word <code>jack</code>
birthday:[1870/01/01 TO 1920/01/01]	Have the <code>birthday</code> values between the specified values. Note that it is a lexicography range

The default search can be controlled using the `Compass::Core` configuration parameters and defaults to all meta-data.

### 10.5.2. CompassHits, CompassDetachedHits & CompassHitsOperations

All the search results are accessible using the `CompassHits` interface. It provides an efficient access to the search results and will only hit the index for "hit number N" when requested. Results are ordered by relevance (if no sorting is provided), in other words and by how well each resource matches the query.

`CompassHits` can only be used within a transactional context, if hits are needed to be accessed outside of a transactional context (like in a jsp view page), they have to be "detached", using one of `CompassHits#detch` methods. The detached hits are of type `CompassDetachedHits`, and it is guaranteed that the index will not be accessed by any operation of the detached hits. `CompassHits` and `CompassDetachedHits` both share the same operations interface called `CompassHitsOperations`.

The following table lists the different `CompassHitsOperations` methods (note that there are many more, please view the javadoc):

**Table 10.2.**

Method	Description
<code>getLength()</code> or <code>length()</code>	Number of resources in the hits.
<code>score(n)</code>	Normalized score (based on the score of the topmost resource) of the n'th top-scoring resource. Guaranteed to be greater than 0 and less than or equal to 1.
<code>resource(n)</code>	Resource instance of the n'th top-scoring resource.
<code>data(n)</code>	Object instance of the n'th top-scoring resource.

### 10.5.3. CompassQuery and CompassQueryBuilder

Compass::Core comes with the `CompassQueryBuilder` interface, which provides programmatic API for building a query. The query builder creates a `CompassQuery` which can then be used to add sorting and executing the query.

Using the `CompassQueryBuilder`, simple queries can be created (i.e. eq, between, prefix, fuzzy), and more complex query builders can be created as well (such as a boolean query, multi-phrase, and query string).

The following code shows how to use a query string query builder and using the `CompassQuery` add sorting to the result.

```
CompassHits hits = session.createQueryBuilder()
    .queryString("+name:jack +familyName:london")
    .setAnalyzer("an1") // use a different analyzer
    .toQuery()
    .addSort("familyName", CompassQuery.SortPropertyType.STRING)
    .addSort("birthdate", CompassQuery.SortPropertyType.INT)
    .hits();
```

Another example for building a query that requires the name to be jack, and the familyName not to be london:

```
CompassQueryBuilder queryBuilder = session.createQueryBuilder();
CompassHits hits = queryBuilder.bool()
    .addMust( queryBuilder.term("name", "jack") )
    .addMustNot( queryBuilder.term("familyName", "london") )
    .toQuery()
    .addSort("familyName", CompassQuery.SortPropertyType.STRING)
    .addSort("birthdate", CompassQuery.SortPropertyType.INT)
    .hits();
```

Note that sorted resource properties / meta-data must be stored and un\_tokenized. Also sorting requires more memory to keep sorting properties available. For numeric types, each property sorted requires four bytes to be cached for each resource in the index. For string types, each unique term needs to be cached.

When a query is built, most of the queries can accept an Object as a parameter, and the name part can be more than just a simple string value of the meta-data / resource-property. If we take the following mapping for example:

```
<class name="eg.A" alias="a">
  <id name="id" />

  <property name="familyName">
    <meta-data>family-name</meta-data>
  </property>

  <property name="date">
    <meta-data converter-param="YYYYMMDD">date-sem</meta-data>
  </property>
```

```
</class>
```

The mapping defines a simple class mapping, with a simple string property called `familyName` and a date property called `date`. With the `CompassQueryBuilder`, most of the queries can directly work with either level of the mappings. Here are some samples:

```
CompassQueryBuilder queryBuilder = session.createQueryBuilder();
// The following search will result in matching "london" against "familyName"
CompassHits hits = queryBuilder.term("a.familyName.family-name", "london").hits();

// The following search will use the class property meta-data id, which in this case
// is the first one (family-name). If there was another meta-data with the family-name value,
// the internal meta-data that is created will be used ($/a/familyName).
CompassHits hits = queryBuilder.term("a.familyName", "london").hits();

// Here, we provide the Date object as a parameter, the query builder will use the
// converter framework to convert the value (and use the given parameter)
CompassHits hits = queryBuilder.term("a.date.date-sem", new Date()).hits();

// Remember, that the alias constraint will not be added automatically, so
// the following query will cause only family-name with the value "london" of alias "a"
CompassHits hits = queryBuilder.bool()
    .addMust( queryBuilder.alias("a") )
    .addMust( queryBuilder.term("a.familyName", "london") )
    .toQuery().hits();
```

### 10.5.4. CompassHighlighter

`Compass::Core` comes with the `CompassHighlighter` interface. It provides ways to highlight matched text fragments based on a query executed. The following code fragment shows a simple usage of the highlighter functionality (please consult the javadoc for more information):

```
CompassHits hits = session.find("london");
// a fragment highlighted for the first hit, and the description property name
String fragment = hits.highlighter(0).fragment("description");
```

Highlighting can only be used with `CompassHits`, which operations can only be used within a transactional context. When working with pure hits results, `CompassHits` can be detached, and then used outside of a transactional context, the question is: what can be done with highlighting?

Each highlighting operation (as seen in the previous code) is also cached within the hits object. When detaching the hits, the cache is passed to the detached hits, which can then be used outside of a transaction. Here is an example:

```
CompassHits hits = session.find("london");
for (int i = 0; i < 10; i++) {
    hits.highlighter(i).fragment("description"); // this will cache the highlighted fragment
}
CompassHit[] detachedHits = hits.detach(0, 10).getHits();

// outside of a transaction (maybe in a view technology)
for (int i = 0; i < detachedHits.length; i++) {
    // this will return the first fragment
    detachedHits[i].getHighlightedText().getHighlightedText();
    // this will return the description fragment, note that the implementation
    // implements the Map interface, which allows it to be used simply in JSTL env and others
    detachedHits[i].getHighlightedText().getHighlightedText("description");
}
```

---

# Part II. Compass::Vocabulary

Compass::Vocabulary aim is to provide common semantic meta-data based on several open forums for online meta-data standards (such as the [Dublin Core Meta data initiative](#)).

---

# Chapter 11. Introduction

Compass::Vocabulary aim is to provide common semantic meta-data based on several open forums for online meta-data standards (such as the [Dublin Core Meta data initiative](#)).

Built on top of the general support for common meta-data, provided by Compass::Core, Compass::Vocabulary provides both a set of common meta data xml definitions files (\*.cmd.xml) and the compiled Java version of them (using the common meta-data ant task).

---

## Chapter 12. Dublin Core

The Compass::Vocabulary supports the [Dublin Core Meta data initiative](#). The common meta data xml mapping files can be found at: `org/compass/vocabulary/dublinCore.cmd.xml`. The generated classes are `org.compass.vocabulary.DublinCore` and `org.compass.vocabulary.DublinCoreOthers`. These classes can be used in your application to use the static String values of the vocabulary.

---

## Part III. Compass::Gps

One of the aims of Compass::Gps is to provide a common API for integrating multiple different indexable data sources (which we are calling Gps devices). An indexable data source could be a file system, ftp site, web page or a database (either via JDBC or ORM tool). A datasource accessed as a GPS device provides the ability to index it's data, either via batch mode or through real time data changes which are mirrored in the index.

Compass::Gps provides an API for registering GPS devices and controlling their lifecycle, along with a set of base classes that implement popular data accessing technologies (i.e JDBC, JDO, Hibernate ORM and OJB). Developers can create their own GPS Device's simply, extending the capability of Compass::Gps.

---

# Chapter 13. Introduction

## 13.1. Overview

Compass::Gps provides integration with different indexable data sources using two concepts: *CompassGps* and *CompassGpsDevice*. Both the interfaces are very abstract, since different data sources are usually different in the way they work or the API they expose.

A device is considered to be any type of indexable data source imaginable, from a database (maybe through the use of an ORM mapping tool), file system, ftp site, or a web site.

The main contract that a device is required to provide is the ability to index it's data (using the `index()` operation). You can think of it as batch indexing the datasource data, providing access for future search queries. An additional possible operation that a device can implement is mirror data changes, either actively or passively.

When performing the index operation, it is very important NOT to perform it within an already running transaction. For `LocalTransactionFactory`, no outer `LocalTransaction` should be started. For `JTATransactionFactory`, no JTA transaction must be started, or no CMT transaction defined for the method level (on EJB Session Bean for example). For `SpringSyncTransactionFactory`, no spring transaction should be wrapping the index code, and the executing method should not be wrapped with a transaction (using transaction proxy for example).

Compass::Gps is built on top of Compass::Core, utilizing all it's features such as transactions (including the important `batch_insert` level for batch indexing), OSEM, and the simple API that comes with Compass::Core.

## 13.2. CompassGps

`CompassGps` is the main interface of `Compass::Gps`. It holds a list of `CompassGpsDevices`, and manages their lifecycle.

`CompassGpsInterfaceDevice` is an extension of `CompassGps`, and provides the needed abstraction between the `Compass` instance/s and the given devices. Every implementation of a `CompassGps` must also implement the `CompassGpsInterfaceDevice`. `Compass::Gps` comes with two implementations of `CompassGps`:

### 13.2.1. SingleCompassGps

Holds a single `Compass` instance. The `Compass` instance is used for both the index operation and the mirror operation. When executing the index operation, the implementation will not use the configured transaction isolation, but will use the `indexTransactionIsolation` transaction isolation property, which defaults to `batch_insert`. You can also set additional settings to be used with the cloned index compass (a prime example is when using jdbc storage, but defining a file location for the indexing compass, thus gaining performance improvement).

When calling the index operation on the `SingleCompassGps`, it will gracefully replace the current index (pointed by the initialized single `Compass` instance), with the content of the index operation. Gracefully means that while the index operation is executing and building a temporary index, no write operations will be allowed on the actual index, and while the actual index is replaced by the temporary index, no read operations are allowed as well.

## 13.2.2. DualCompassGps

Holds two `Compass` instances. One, called `indexCompass` is responsible for index operation. The other, called `mirrorCompass` is responsible for mirror operations. The main reason why we have two different instances is because the transaction isolation level can greatly affect the performance of each operation. Usually the `indexCompass` instance will be configured with the `batch_insert` isolation level, while the `mirrorCompass` instance will use the default transaction isolation level (`read_committed`).

When calling the index operation on the `DualCompassGps`, it will gracefully replace the mirror index (pointed by the initialized `mirrorCompass` instance), with the content of the index `index` (pointed by the initialized `indexCompass` instance). Gracefully means that while the index operation is executing and building the index, no write operations will be allowed on the mirror index, and while the mirror index is replaced by the index, no read operations are allowed as well.

Both implementations of `CompassGps` allow to set / override settings of the `Compass` that will be responsible for the index process. One sample of using the feature which might yield performance improvements can be when storing the index within a database. The indexing process can be done on the local file system (on a temporary location), in a compound format (or non compound format), by setting the indexing compass connection setting to point to a file system location. Both implementations will perform "hot replace" of the file system index into the database location, automatically compounding / uncompounding based on the settings of both the index and the mirror compass instances.

## 13.3. CompassGpsDevice

A Gps devices must implement the `CompassGpsDevice` interface in order to provide device indexing. It is responsible for interacting with a data source and reflecting it's data in the `Compass` index. Two examples of devices are a file system and a database, accessed through the use of a ORM tool (like Hibernate).

A device will provide the ability to index the data source (using the `index()` operation), which usually means iterating through the device data and indexing it. It might also provide "real time" monitoring of changes in the device, and applying them to the index as well.

A `CompassGpsDevice` cannot operate standalone, and must be a part of a `CompassGps` instance (even if we have only one device), since the device requires the `Compass` and instance(s) in order to apply the changes to the index.

Each device has a name associated with it. A device name must be unique across all the devices within a single `CompassGps` instance.

### 13.3.1. MirrorDataChangesGpsDevice

As mentioned, the main operation in `CompassGpsDevice` is `index()`, which is responsible for batch indexing all the relevant data in the data source. Gps devices that can mirror real time data changes made to the data source by implementing the `MirrorDataChangesGpsDevice` interface (which extends the `CompassGpsDevice` interface).

There are two types of devices for mirroring data. `ActiveMirrorGpsDevice` provides data mirroring of the datasource by explicit programmatic calls to `performMirroring`. `PassiveMirrorGpsDevice` is a GPS device that gets notified of data changes made to the data source, and does not require user intervention in order to reflect data changes to the compass index.

For `ActiveMirrorGpsDevice`, `Compass::Gps` provides a `ScheduledMirrorGpsDevice` class, which wraps an `ActiveMirrorGpsDevice` and schedules the execution of the `performMirror()` operation.

## 13.4. Programmatic Configuration

Configuration of `Compass::Gps` is achieved by programmatic configuration or through an IOC container. All the devices provided by `Compass::Gps` as well as `CompassGps` can be configured via Spring framework.

The following code snippet shows how to configure `Compass::Gps` as well as managing it's lifecycle.

```
Compass compass = ... // configure compass
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice device1 = ... // configure the first device
device1.setName("device1");
gps.addDevice(device1);
CompassGpsDevice device2 = ... // configure the second device
device2.setName("device2");
gps.addDevice(device2);

gps.start();
....
....
//on application shutdown
gps.stop();
```

## 13.5. Building a Gps Device

If you wish to build your own Gps Device, it could not be simpler (actually, it is as simple as getting the data from the data source or monitoring the data source data changes). The main API that a device must implement is `index()` which by contract means that all the relevant data for indexing in the data source is indexed.

If you wish to perform real time mirroring of data changes from the data source to the index, you can control the lifecycle of the mirroring using the `start()` and `stop()` operations, and must implement either the `ActiveMirrorGpsDevice` or the `PassiveMirrorGpsDevice` interfaces.

`Compass::Gps` comes with a set of base classes for gps devices that can help the development of new gps devices.

---

# Chapter 14. JDBC

## 14.1. Introduction

The Jdbc Gps Device provides support for database indexing through the use of JDBC. The Jdbc device maps a Jdbc ResultSet to a set of Compass Resources (sharing the same resource mapping). Each Resource maps one to one with a ResultSet row. The Jdbc device can hold multiple ResultSet to Resource mappings. The Jdbc Gps device class is ResultSetJdbcGpsDevice. The core configuration is the mapping definitions of a Jdbc ResultSet and a Compass Resource.

The Jdbc Gps device does not use OSEM, since no POJOs are defined that map the ResultSet to objects. For applications that use ORM tools, Compass::Gps provides several devices that integrate with popular ORM tools such as Hibernate, JDO, and OJB. For more information about Compass Resource, Resource Property and resource mapping, please read the Search Engine and Resource Mapping sections.

The Jdbc Gps device also provides support for ActiveMirrorGpsDevice, meaning that data changes done to the database can be automatically detected by the defined mappings and device.

For the rest of the chapter, we will use the following database tables:

```
CREATE TABLE parent (
  id INTEGER NOT NULL IDENTITY PRIMARY KEY,
  first_name VARCHAR(30),
  last_name VARCHAR(30),
  version BIGINT NOT NULL
);
CREATE TABLE child (
  id INTEGER NOT NULL IDENTITY PRIMARY KEY,
  parent_id INTEGER NOT NULL,
  first_name VARCHAR(30),
  last_name VARCHAR(30),
  version BIGINT NOT NULL
);
alter table child add constraint
  fk_child_parent foreign key (parent_id) references parent(id);
```

The PARENT.ID is the primary key of the PARENT table, and the CHILD.ID is the primary key of the CHILD table. There is a one to many relationship between PARENT and child using the CHILD.PARENT\_ID column. The VERSION columns will be explained later, as they are used for the data changes mirroring option.

## 14.2. Mapping

To enable the Jdbc device to index a database, a set of mappings must be defined between the database and the compass index. The main mapping definition maps a generic Jdbc ResultSet to a set of Compass Resources that are defined by a specific Resource Mapping definitions. The mapping can be configured either at database ResultSet or Table levels. ResultSetToResourceMapping maps generic select SQL (returning a ResultSet) and TableToResourceMapping (extends the ResultSetToResourceMapping) simply maps database tables.

### 14.2.1. ResultSet Mapping

The following code sample shows how to configure a single ResultSet that combines both the PARENT and CHILD tables into a single resource mapping with an alias called "result-set".

```
ResultSetToResourceMapping mapping = new ResultSetToResourceMapping();
```

```

mapping.setAlias("result-set");
mapping.setSelectQuery("select "
    + "p.id as parent_id, p.first_name as parent_first_name, p.last_name as parent_last_name, "
    + "c.id as child_id, c.first_name as child_first_name, c.last_name child_last_name "
    + "from parent p left join child c on p.id = c.parent_id");
// maps from a parent_id column to a resource property named parent-id
mapping.addIdMapping(new IdColumnToPropertyMapping("parent_id", "parent-id"));
// maps from a child_id column to a resource property named child-id
mapping.addIdMapping(new IdColumnToPropertyMapping("child_id", "child-id"));
mapping.addDataMapping(new DataColumnToPropertyMapping("parent_first_name", "parent-first-name"));
mapping.addDataMapping(new DataColumnToPropertyMapping("parent_first_name", "first-name"));
mapping.addDataMapping(new DataColumnToPropertyMapping("child_first_name", "child-first-name"));
mapping.addDataMapping(new DataColumnToPropertyMapping("child_first_name", "first-name"));

```

Here, we defined a mapping from a `ResultSet` that combines both the `PARENT` table and the `CHILD` table into a single set of `Resources`. Note also in the above example how "parent\_first\_name" is mapped to multiple alias names, allowing searches to be performed on either the specific attribute type or the more general "first\_name".

The required settings for the `ResultSetToResourceMapping` are the alias name of the `Resource` that will be created, the select query that generates the `ResultSet`, and the ids columns mapping (at least one must be defined) that maps to the columns the uniquely identifies the rows in the `ResultSet`.

`ColumnToPropertyMapping` is a general mapping from a database column to a `Compass Resource Property`. The mapping can map from a column name or a column index (the order that it appears in the select query) to a `Property` name. It can also have definitions of the `Property` characteristics (`Property.Index`, `Property.Store` and `Property.TermVector`). Both `IdColumnToPropertyMapping` and `DataColumnToPropertyMapping` are of `ColumnToPropertyMapping` type.

In the above sample, the two columns that identifies a row for the given select query, are the `parent_id` and the `child_id`. They are mapped to the `parent-id` and `child-id` property names respectively.

Mapping data columns using the `DataColumnToPropertyMapping` provides mapping from "data" columns into searchable meta-data (`Resource Property`). As mentioned, you can control the property name and its characteristics. Mapping data columns is optional, though mapping none makes little sense. `ResultSetToResourceMapping` has the option to index all the unmapped columns of the `ResultSet` by setting the `indexUnMappedColumns` property to `true`. The meta-datas that will be created will have the property name set to the column name.

## 14.2.2. Table Mapping

`TableToResourceMapping` is a simpler mapping that extends the `ResultSetToResourceMapping`, and maps a database table to a resource mapping. The following code sample shows how to configure the table mapping.

```

TableToResourceMapping parentMapping = new TableToResourceMapping("PARENT", "parent");
parentMapping.addDataMapping(new DataColumnToPropertyMapping("first_name", "first-name"));
TableToResourceMapping childMapping = new TableToResourceMapping("CHILD", "child");
childMapping.addDataMapping(new DataColumnToPropertyMapping("first_name", "first-name"));

```

The above code defined the table mappings. One mapping for the `PARENT` table to the "parent" alias, and one for the `CHILD` table to the "child" alias. The mappings definitions are much simpler than the `ResultSetToResourceMapping`, with only the table name and the alias required. Since the mapping works against a database table, the id columns can be auto generated (based on the table primary keys, and the property names are the same as the column names), and the select query (based on the table name). Note that the mapping will auto generate only settings that have not been set. If for example the select query is set, it will not be generated.

## 14.3. Mapping - MirrorDataChanges

The `ResultSetJdbcGpsDevice` supports mirroring data changes to the database. In order to enable it, the `ResultSet` that will be mapped must have at least one version column. The version column must be incremented whenever a change occurs to the corresponding row in the database (Note that some databases have the feature built in, like ORACLE).

### 14.3.1. ResultSet Mapping

The following code sample shows how to configure a mirroring enabled `ResultSet` mapping:

```
ResultSetToResourceMapping mapping = new ResultSetToResourceMapping();
mapping.setAlias("result-set");
mapping.setSelectQuery("select "
    + "p.id as parent_id, p.first_name as parent_first_name, p.last_name as parent_last_name, p.version as parent_
    + "COALESCE(c.id, 0) as child_id, c.first_name as child_first_name, c.last_name child_last_name, COALESCE(c.v
    + "from parent p left join child c on p.id = c.parent_id");
mapping.setVersionQuery("select p.id as parent_id, COALESCE(c.id, 0) as child_id, "
    + "p.version as parent_version, COALESCE(c.version, 0) as child_version "
    + "from parent p left join child c on p.id = c.parent_id");
mapping.addIdMapping(new IdColumnToPropertyMapping("parent_id", "parent-id", "p.id"));
mapping.addIdMapping(new IdColumnToPropertyMapping("child_id", "child-id", "COALESCE(c.id, 0)"));
mapping.addDataMapping(new DataColumnToPropertyMapping("parent_first_name", "parent-first-name"));
mapping.addDataMapping(new DataColumnToPropertyMapping("child_first_name", "child-first-name"));
mapping.addVersionMapping(new VersionColumnMapping("parent_version"));
mapping.addVersionMapping(new VersionColumnMapping("child_version"));
```

There are three additions to the previously configured result set mapping. The first is the version query that will be executed in order to identify changes made to the result set (rows created, updated, or deleted). The version query should return the `ResultSet` id and version columns. The second change is the id columns names in the select query, since a dynamic where clause is added to the select query for mirroring purposes. The last one is the actual version column mapping (no version column mapping automatically disabled the mirroring feature).

### 14.3.2. Table Mapping

The following code sample shows how to configure a mirroring enabled Table mapping:

```
TableToResourceMapping parentMapping = new TableToResourceMapping("parent", "parent");
parentMapping.addVersionMapping(new VersionColumnMapping("version"));
parentMapping.setIndexUnMappedColumns(true);

TableToResourceMapping childMapping = new TableToResourceMapping("child", "child");
childMapping.addVersionMapping(new VersionColumnMapping("version"));
childMapping.setIndexUnMappedColumns(true);
```

Again, the table mapping is much simpler than the result set mapping. The only thing that needs to be added is the version column mapping. The version query is automatically generated.

### 14.3.3. Jdbc Snapshot

The mirroring operation works with snapshots. Snapshots are taken when the `index()` or the `performMirroring()` are called and represents the latest `ResultSet` state.

`Compass::Gps` comes with two snapshot mechanisms. The first is `JdbcSnapshotPersister:RAMJdbcSnapshotPersister` which holds the `Jdbc` snapshot in memory and is not persistable between application lifecycle. The second is `FSJdbcSnapshotPersister`, which save the snapshot in the file system

(using the given file path). A code sample:

```
gpsDevice = new ResultSetJdbcGpsDevice();
gpsDevice.setSnapshotPersister(new FSJdbcSnapshotPersister("target/testindex/snapshot"));
```

## 14.4. Resource Mapping

After defining the result set mapping, the resource mapping must be defined as well. Luckily, there is no need to create the mapping file (cpm.xml file), since it can be generated automatically using `Compass::Core MappingResolver` feature. The `Jdbc` device provides the `ResultSetResourceMappingResolver` which automatically generates the resource mapping for a given `ResultSetToResourceMapping`. Additional settings for the resource mapping can be set as well, such as the sub-index, all meta data, etc.

```
CompassConfiguration conf = new CompassConfiguration()
    .setSetting(CompassEnvironment.CONNECTION, "target/testindex");

DataSource dataSource = // get/create a Jdbc Data Source
ResultSetToResourceMapping mapping = // create the result set mapping

conf.addMappingResover(new ResultSetResourceMappingResolver(mapping, dataSource));
```

## 14.5. Putting it All Together

After explaining two of the most important aspects of the `Jdbc` mappings, here is a complete example of configuring a `ResultSetJdbcGpsDevice`.

```
ResultSetToResourceMapping mapping1 = // create the result set mapping or table mapping
ResultSetToResourceMapping mapping2 = // create the result set mapping or table mapping
DataSource dataSource = // create a jdbc dataSource or look it up from JNDI

CompassConfiguration conf = new CompassConfiguration().setSetting(CompassEnvironment.CONNECTION,
"target/testindex");
conf.addMappingResover(new ResultSetResourceMappingResolver(mapping1, dataSource));

// build the mirror compass instance
compass = conf.buildCompass();

gpsDevice = new ResultSetJdbcGpsDevice();
gpsDevice.setDataSource(dataSource);
gpsDevice.setName("jdbcDevice");
gpsDevice.setMirrorDataChanges(false);
gpsDevice.addMapping(mapping1);
gpsDevice.addMapping(mapping2);

gps = new SingleCompassGps(compass);
gps.addGpsDevice(gpsDevice);
gps.start();
```

GPS devices are Inversion Of Control / Dependency Injection enabled, meaning that it can be configured with an IOC container. For an example of configuring the `ResultSetJdbcGpsDevice`, please see [Spring Jdbc Gps Device](#) section.

---

# Chapter 15. Hibernate

## 15.1. Introduction

The Hibernate Gps Device provides support for database indexing through the use of [Hibernate](#) ORM mappings. If your application uses Hibernate, it couldn't be easier to integrate Compass into your application (Sometimes with no code attached - see the petclinic sample).

Hibernate Gps Device utilizes Compass::Core OSEM feature (Object to Search Engine Mappings) and Hibernate ORM feature (Object to Relational Mappings) to provide simple database indexing. As well as Hibernate 3 new event based system to provide real time mirroring of data changes done through Hibernate. The path data travels through the system is: Database -- Hibernate -- Objects -- Compass::Gps -- Compass::Core (Search Engine).

## 15.2. Configuration

When configuring the Hibernate device, one must instantiate either `Hibernate2GpsDevice` (for Hibernate 2 version) or `Hibernate3GpsDevice` (for Hibernate 3 version). After instantiating the device, it must be initialized by either a `Hibernate Configuration` or a `Hibernate SessionFactory`. When configuring the device with `Hibernate Configuration`, a new `SessionFactory` is created when the device is started.

It is more preferable to configure the device with the `SessionFactory` that the actual application will use, especially since data mirroring will only work when both the device and the application will use the same `SessionFactory`.

Here is a code sample of how to configure the hibernate device:

```
Compass compass = ... // set compass instance
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice hibernateDevice =
// If Hibernate 2
    new Hibernate2GpsDevice("hibernate", sessionFactory);
// If Hibernate 3
    new Hibernate3GpsDevice("hibernate", sessionFactory);
gps.addDevice(hibernateDevice);
... // configure other devices
gps.start();
```

## 15.3. Index Operation

Hibernate Gps device provides the ability to index a database. It supports both Hibernate 2 and Hibernate 3 versions. Compass will index objects (or their matching database tables in the Hibernate mappings) specified in both the Hibernate mappings and Compass::Core mappings (OSEM) files.

When indexing Compass::Gps, the Hibernate device can be configured with a `fetchCount`. The `fetchCount` parameter controls the pagination process of indexing a class (and it's represented table) so in case of large tables, the memory level can be controlled.

## 15.4. Real Time Data Mirroring

The Hibernate Gps Device, with Hibernate 3 new event system, provides support for real time data mirroring. Data changes via Hibernate are reflected in the Compass index. There is no need to configure anything in order to enable the feature, the device takes care for it all.

An important point when configuring the hibernate device is that both the application and the hibernate device must use the same `SessionFactory`. Which means that the device must be configured with a `SessionFactory` and not a `Configuration`.

Note on Hibernate 2 and `Interceptors`. When using generated ids with Hibernate 2, the id in the interceptor is `null`, which means that when creating new objects and persisting them to the database, the device has no way to index the object. If Hibernate 2 is a must, one possible solution is to use aspects.

If using Hibernate 3 and the Spring Framework, please see the `SpringHibernate3GpsDevice`

---

# Chapter 16. JPA (Java Persistence API)

## 16.1. Introduction

The Jpa Gps Device provides support for database indexing through the use of the Java Persistence API (Jpa), part of the EJB3 standard. If your application uses Jpa, it couldn't be easier to integrate Compass into your application.

Jpa Gps Device utilizes Compass::Core OSEM feature (Object to Search Engine Mappings) and Jpa feature (Object to Relational Mappings) to provide simple database indexing. As well as Jpa support for life-cycle event based system to provide real time mirroring of data changes done through Jpa (see notes about real time mirroring later on). The path data travels through the system is: Database -- Jpa (Entity Manager) -- Objects -- Compass::Gps -- Compass::Core (Search Engine).

## 16.2. Configuration

When configuring the Jpa device, one must instantiate `JpaGpsDevice`. After instantiating the device, it must be initialized by an `EntityManagerFactory`. This is the only required parameter to the `JpaGpsDevice`. For tighter integration with the actual implementation of Jpa (i.e. Hibernate), and frameworks that wrap it (i.e. Spring), the device allows for abstractions on top of it. Each one will be explained in the next sections, though in the spirit of compass, it already comes with implementations for popular Jpa implementations.

Here is a code sample of how to configure the Jpa device:

```
Compass compass = ... // set compass instance
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice jpaDevice =
    new JpaGpsDevice("jpa", entityManagerFactory);
gps.addDevice(jpaDevice);
... // configure other devices
gps.start();
```

The device performs all its operations using its `EntityManagerWrapper`. The Jpa support comes with three different implementations: `JtaEntityManagerWrapper` which will only work within a JTA environment, `ResourceLocalEntityManagerWrapper` for resource local transactions, and `DefaultEntityManagerWrapper` which works with both JTA and resource local environments. The `DefaultEntityManagerWrapper` is the default implementation of the `EntityManagerWrapper` the device will use.

Several frameworks (like Spring) sometimes wrap (proxy) the actual `EntityManagerFactory`. Some features of the Jpa device require the actual implementation of the `EntityManagerFactory`. This features are the ones that integrate tightly with the implementation of the `EntityManagerFactory`, which are described later in the chapter. The device allows to set `NativeEntityManagerFactoryExtractor`, which is responsible for extracting the actual implementation.

## 16.3. Index Operation

Jpa Gps device provides the ability to index a database. It automatically supports all different Jpa implementations. Compass will index objects (or their matching database tables in the Jpa mappings) specified in both the Jpa mappings and Compass::Core mappings (OSEM) files.

When indexing `Compass::Gps`, the `Jpa` device can be configured with a `fetchCount`. The `fetchCount` parameter controls the pagination process of indexing a class (and its represented table) so in case of large tables, the memory level can be controlled.

The device allows to set a `JpaEntitiesLocator`, which is responsible for extracting all the entities that are mapped in both `Compass` and `Jpa EntityManager`. The default implementation `DefaultJpaEntitiesLocator` uses `Annotations` to determine if a class is mapped to the database. Most of the times, this will suffice, but for applications that use both annotations and `xml` definitions, a tighter integration with the `Jpa` implementation is required, with a specialized implementation of the locator. `Compass` comes with several specialized implementations of a locator, and auto-detect the one to use (defaulting to the default implementation if none is found). Note, that this is one of the cases where the actual `EntityManagerFactory` is required, so if the application is using a framework that wraps the `EntityManagerFactory`, a `NativeEntityManagerFactoryExtractor` should be provided.

## 16.4. Real Time Data Mirroring

The `Jpa` specification allows for declaring life-cycle event listeners either on the actual domain model using annotations, or in the persistence settings. The `EntityManagerFactory` API does not allow for a way to register global listeners programatically. `Compass` comes with two abstract support classes to ease the definition of listeners. The first is the `AbstractCompassJpaEntityListener`, which requires the implementation to implement the `getCompass` which will fetch the actual `compass` instance (probably from `Jndi`). The second is the `AbstractDeviceJpaEntityListener`, which requires the implementation to implement the `getDevice` which will fetch the `Jpa Gps Device`.

With several `Jpa` implementation, `Compass` can automatically register life-cycle event listeners based on the actual implementation API's (like `Hibernate` event listeners support). In order to enable it, the `injectEntityLifecycleListener` must be set to `true` (defaults to `false`), and an implementation of `JpaEntityLifecycleInjector` can be provided. `Compass` can auto-detect a proper injector based on the currently provided internal injector implementations. The auto-detection will happen if no implementation for the injector is provided, and the `inject` flag is set to `true`. Note, that this is one of the cases where the actual `EntityManagerFactory` is required, so if the application is using a framework that wraps the `EntityManagerFactory`, a `NativeEntityManagerFactoryExtractor` should be provided.

An important point when configuring the `Jpa` device is that both the application and the `Jpa` device must use the same `EntityManagerFactory`.

---

# Chapter 17. JDO (Java Data Objects)

## 17.1. Introduction

The Jdo Gps Device provides support for database indexing through the use of Jdo ORM mappings. If your application uses Jdo, it couldn't be easier to integrate Compass into your application.

Jdo Gps Device utilizes Compass::Core OSEM feature (Object to Search Engine Mappings) and Jdo ORM feature (Object to Relational Mappings) to provide simple database indexing. As well as Jdo 2 new event based system to provide real time mirroring of data changes done through the Jdo 2 implementation. The path data travels through the system is: Database -- Jdo -- Objects -- Compass::Gps -- Compass::Core (Search Engine).

## 17.2. Configuration

When configuring the Jdo device, one must instantiate either `JdoGpsDevice` (for Jdo 1 version) or `Jdo2GpsDevice` (for Jdo 2 version). After instantiating the device, it must be initialized `JdoPersistenceManagerFactory`.

Here is a code sample of how to configure the hibernate device:

```
Compass compass = // set compass instance
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice jdoDevice =
// If Jdo 1
    new JdoGpsDevice("jdo", persistenceManagerFactory);
// If Jdo 2
    new Jdo2GpsDevice("jdo", persistenceManagerFactory);
gps.addDevice(jdoDevice);
... // configure other devices
gps.start();
```

## 17.3. Index Operation

Jdo Gps device provides the ability to index the database. It supports both Jdo and Jdo 2 versions. Compass will index objects (or their matching database tables in the Jdo mappings) specified in both the Jdo mappings and Compass::Core mappings (OSEM) file.

## 17.4. Real Time Data Mirroring

The Jdo 2 Gps Device, with Jdo 2 new event system, provides support for real time data mirroring. Data changes via Jdo are reflected in the Compass index. There is no need to configure anything in order to enable the feature, the device takes care for it all.

An important point when configuring the jdo device is that both the application and the jdo device must use the same `PersistenceManagerFactory`.

---

# Chapter 18. OJB (Object Relational Broker)

## 18.1. Introduction

The OJB Gps Device provides support for database indexing through the use of Apache OJB ORM mappings. If your application uses OJB, it couldn't be easier to integrate Compass into your application (Sometimes with no code attached - see the petclinic sample).

OJB Device uses Compass::Core OSEM feature (Object to Search Engine Mappings) and OJB ORM feature (Object to Relational Mappings) to provide simple database indexing. The device also utilizes OJB lifecycle events to provide real time mirroring of data changes done through OJB. The path data travels through the system is: Database -- OJB -- Objects -- Compass::Gps -- Compass::Core (Search Engine).

## 18.2. Index Operation

OJB device provides the ability to index the database. The objects that will be indexed (or their matching database tables in the OJB mappings) are ones that have both OJB mappings and Compass::Core mappings (OSEM).

Indexing the data (using the

`index()`

operation) requires the `indexPersistentBroker` property to be set, before the `index()` operation is called. You can use the `OjbGpsDevice#attachPersistenceBrokerForIndex(CompassGpsDevice, PersistenceBroker)` as a helper method.

## 18.3. Real Time Data Mirroring

Real-time mirroring of data changes requires using the `OjbGpsDevice#attachLifecycleListeners(PersistenceBroker)` to let the device listen for any data changes, and `OjbGpsDevice#removeLifecycleListeners(PersistenceBroker)` to remove the listener. Since the lifecycle listener can only be set on the instance level and not the factory level, attach and remove must be called every time a `PersistentBroker` is instantiated. You can use the `OjbGpsDeviceUtils#attachPersistentBrokerForMirror(CompassGpsDevice, PersistenceBroker)` and `OjbGpsDeviceUtils#removePersistentBrokerForMirror(CompassGpsDevice, PersistenceBroker)` as helper methods if attachment/removal is required for a generic device (i.e. `OjbGpsDevice`).

Since the real time mirroring and the event listener registration sounds like an aspect for `Ojb` aware classes/methods, `Compass::Spring` utilizes spring support for OJB and aspects for a much simpler event registration, please see `Compass::Spring` for more documentation.

## 18.4. Configuration

Here is a code sample of how to configure the ojb device:

```
Compass compass = ... // set compass instance
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice ojbDevice = new OjbGpsDevice();
ojbDevice.setName("ojb");
gps.addDevice(ojbDevice);
```

```
.... // configure other devices
gps.start();
....
// just before calling the index method
PersistenceBroker pb = // create Persistence Broker
OjbGpsDeviceUtils.attachPersistenceBrokerForIndex(objDevice, pb);
gps.index();
....
// a Persistence Broker operation level
PersistenceBroker pb = // create Persistence Broker
OjbGpsDeviceUtils.attachPersistenceBrokerForMirror(objDevice, pb);
.... // Persistence Broker operations
OjbGpsDeviceUtils.removePersistenceBrokerForMirror(objDevice, pb);
```

---

# Chapter 19. iBatis

## 19.1. Introduction

The `SqlMapClient` ([iBatis](#)) Gps Device provides support for database indexing through the use of Apache iBatis ORM mappings. The device can index the database data using a set of configured select statements. Mirroring is not supported, but if Spring is used, `Compass::Spring AOP` can be simply used to add advices that will mirror data changes that are made using iBatis DAOs.

## 19.2. Index Operation

When using iBatis and it's `SqlMapClient`, the application has several `sqlMap` configuration files. The `sqlMap` configuration usually holds configuration for a specific class, and holds it's respective insert/update/delete/select operations. The `Compass iBatis` support can use the select statements to fetch the data from the database. When creating the `SqlMapClientGpsDevice`, an array of select statements ids can be supplied, and the device will execute and index all of them. If the selects requires parameters as well, an additional array of Object parameters can be supplied, matching one to one with the select statements.

## 19.3. Configuration

Here is a code sample of how to configure the `SqlMapClient` (iBatis) device:

```
Compass compass = ... // set compass instance
CompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice ibatisDevice = new SqlMapClientGpsDevice();
SqlMapClient sqlMapClient = ... // set sqlMapClient instance
ibatisDevice.setName("ibatis", sqlMapClient, new String[] {"getUsers", "getContacts"});
gps.addDevice(ibatisDevice);
... // configure other devices
gps.start();
....
gps.index();
```

---

# Part IV. Compass::Spring

Compass::Spring aim is to provide a closer integration with the [springframework](#).

---

# Chapter 20. Introduction

## 20.1. Overview

Compass::Spring aim is to provide closer integration with the [springframework](#). The following list summarizes the main integration points with Spring.

- Support for a `Compass` level factory bean, with Spring IOC modelled configuration options.
- Compass DAO level support (similar to the ORM dao support), with transaction integration and Compass DAO support class.
- An extension on top of Spring's Hibernate 3 dao support which extends `Compass::Gps` Hibernate 3 device. Handles Spring proxing of the Hibernate `SessionFactory`.
- An extension on top of Spring's OJB dao support which extends `Compass::Gps` OJB device. Mainly provides non programmatic configuration with OJB.
- Extension to Spring MVC, providing Search controller (based on `Compass::Core` search capabilities) and an Index controller (based on `Compass::Gps` index operation).

## 20.2. Compass Definition in Application Context

Compass::Spring provides the ability to expose `Compass` as a Spring bean from an application context file. Application objects that need to access `Compass` will obtain a reference to a pre-defined instance via bean references. The following is an example of a Spring XML application context definition configuring `Compass`:

```
<beans>
  ...
  <bean id="compass"
    class="org.compass.spring.LocalCompassBean">
    <property name="resourceLocations">
      <list>
        <value>classpath:org/compass/spring/test/A.cpm.xml</value>
      </list>
    </property>
    <property name="compassSettings">
      <props>
        <prop key="compass.engine.connection">
          target/testindex
        </prop>
        <!-- This is the default transaction handling
          (just explicitly setting it) -->
        <prop key="compass.transaction.factory">
          org.compass.core.transaction.LocalTransactionFactory
        </prop>
      </props>
    </property>
  </bean>
  ...
</beans>
```

If using a Spring `PlatformTransactionManager`, you should also initialize the `transactionManager` property

of the `LocalCompassBean`.

Also, of storing the index within a database, be sure to set the `dataSource` property of the `LocalCompassBean`. It will be automatically wrapped by Spring's `TransactionAwareDataSourceProxy` if not wrapped already.

---

# Chapter 21. DAO Support

## 21.1. Dao and Template

Compass::Spring uses the `CompassTemplate` and `CompassCallback` classes provided by `Compass::Core` module as part of its DAO (Data Access Object) support for Spring.

Compass::Spring provides a simple base class called `CompassDaoSupport` which can be initialized by `Compass` or `CompassTemplate` and provides access to `CompassTemplate` from its subclasses.

The following code shows a simple Library Dao:

```
public class LibraryCompassDao extends CompassDaoSupport {
    public int getNumberOfHits(final String query) {
        Integer numberOfHits = (Integer)getCompassTemplate().execute(
            new CompassCallback() {
                public Object doInCompass(CompassSession session) {
                    CompassHits hits = session.find(query);
                    return new Integer(hits.getLength());
                }
            }
        );
    }
    return numberOfHits.intValue();
}
```

The following is an example of configuring the above Library DAO in the XML application context (assuming that we configured a `LocalCompassBean` named "compass" previously):

```
<beans>
  <bean id="libraryCompass" class="LibraryCompassDao">
    <property name="compass">
      <ref local="compass" />
    </property>
  </bean>
</beans>
```

---

# Chapter 22. Spring Transaction

## 22.1. Introduction

Compass::Spring integrates with Spring transaction management in several ways, either using Compass::Core own LocalTransaction or using the Spring transaction synchronization services. Currently there is no Compass implementation of Spring's PlatformTransactionManagement.

## 22.2. LocalTransaction

Compass::Core default transaction handling is LocalTransaction. A LocalTransaction does not integrate with Spring transaction management services, but can be used to write Compass Dao beans that do not require integration with an on going Spring or Jta transactions.

## 22.3. JTASyncTransaction

When using Spring's JtaTransactionManager, you have a choice to either use the SpringSyncTransaction (described next) or the JTASyncTransaction provided by Compass::Core (where SpringSyncTransaction is preferable).

## 22.4. SpringSyncTransaction

Compass::Spring integrates with Spring transaction synchronization services. This means that whichever Spring transaction manager (Jta, Hiberante, ...) you are using, the SpringSyncTransaction will synchronize with the transaction upon transaction completion.

If you are using the SpringSyncTransaction, a Spring based transaction must already be started in order for SpringSyncTransaction to join. If no transaciton is started, Compass can start one (and will commit it eventually) if the PlatformTransactionManager is provided to the LocalCompassBean. The transaction must support the transaction synchronization feature (which by default all of them do).

Note: you can use spring transaction management support to suspend and resumed transactions. In which case a Compass provided transaction will be suspended and resumed also.

In order to configure Compass to work with the SpringSyncTransaction, you must set the `compass.transaction.factory` to `org.compass.spring.transaction.SpringSyncTransactionFactory`.

## 22.5. CompassTransactionManager

Currently Compass::Spring does not provide a CompassTransactionManager. This means any CompassDao objects with LocalTransaction, programmatic (Spring transection template) / declarative (Spring Interceptor/AOP transaction support) Spring transaction definition won't be applied to the Compass transaction.

---

# Chapter 23. Hibernate 3 Gps Device Support

## 23.1. Introduction

The device is built on top of Spring ORM support for Hiberante 3, and Compass::Gps support for Hibernate 3 device. It provides support for Spring generation of Hibernate `SessionFactory` proxy.

## 23.2. SpringHibernate3GpsDevice

An extension of the `Hibernate3GpsDevice` that can handle Spring's proxing the Hibernate `SessionFactory` in order to register event listeners for real time data changes mirroring.

---

# Chapter 24. OJB Gps Device Support

## 24.1. Introduction

Compass OJB support is built on top of Spring ORM support for Apache OJB (Object Relational Broker) and the Compass::Gps support for OJB device. This provides simpler integration with OJB. For a complete and working sample, please see the petclinic sample.

## 24.2. SpringOjbGpsDevice

`SpringOjbGpsDevice` is an extension of the `OjbGpsDevice` and utilizes Spring ojb features. This device Uses Spring `PersistenceBrokerTemplate` and `OjbFactoryUtils` to get the current `PersistenceBroker` for batch indexing (the `index()` operation).

You can provide the `PersistenceBrokerTemplate`, though it is not required since it is created the same way the `PersistenceBrokerDaoSupport` does.

The device can be used with `SpringOjbGpsDeviceInterceptor` to provide real-time data mirroring without the need to write any code (described in the next section).

## 24.3. SpringOjbGpsDeviceInterceptor

`SpringOjbGpsDeviceInterceptor` Uses Spring's AOP capabilities to attach and remove lifecycle event listeners to the `PersistenceBroker` (the device acts as the listener). Uses `OjbGpsDeviceUtils` to perform it on the supplied `SpringOjbGpsDevice`.

Mainly used as a post interceptor with transaction proxies that manage service layer operations on an OJB enabled DAO layer.

---

# Chapter 25. Jdbc Gps Device Support

## 25.1. Introduction

This section provides no additional implementation, only samples of using Jdbc Gps Device within Spring IOC container.

The database structure is the same one as the one on the Jdbc Gps Device section, and is show here as well:

```
CREATE TABLE parent (
  id INTEGER NOT NULL IDENTITY PRIMARY KEY,
  first_name VARCHAR(30),
  last_name VARCHAR(30),
  version BIGINT NOT NULL
);
CREATE TABLE child (
  id INTEGER NOT NULL IDENTITY PRIMARY KEY,
  parent_id INTEGER NOT NULL,
  first_name VARCHAR(30),
  last_name VARCHAR(30),
  version BIGINT NOT NULL
);
alter table child add constraint
  fk_child_parent foreign key (parent_id) references parent(id);
```

## 25.2. ResultSet Mapping

A configuration sample of a the ResultSet mapping given at the Jdbc Gps Device section is shown here in a Spring configuration file (taken from the unit tests):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <bean id="rsMapping" class="org.compass.gps.device.jdbc.mapping.ResultSetToResourceMapping">
    <property name="alias"><value>result-set</value></property>
    <property name="indexUnMappedColumns"><value>true</value></property>
    <property name="selectQuery"><value>
      select  p.id as parent_id,
              p.first_name as parent_first_name,
              p.last_name as parent_last_name,
              p.version as parent_version,
              COALESCE(c.id, 0) as child_id,
              c.first_name as child_first_name,
              c.last_name child_last_name,
              COALESCE(c.version, 0) as child_version
      from
        parent p left join child c on p.id = c.parent_id
    </value></property>
    <property name="versionQuery"><value>
      select  p.id as parent_id,
              COALESCE(c.id, 0) as child_id,
              p.version as parent_version,
              COALESCE(c.version, 0) as child_version
      from
        parent p left join child c on p.id = c.parent_id
    </value></property>
    <property name="idMappings">
      <list>
        <bean class="org.compass.gps.device.jdbc.mapping.IdColumnToPropertyMapping">
          <property name="columnName"><value>parent_id</value></property>
          <property name="propertyName"><value>parent_id</value></property>
          <property name="columnNameForVersion"><value>p.id</value></property>
        </bean>
      </list>
    </property>
  </bean>
```

```

    <bean class="org.compass.gps.device.jdbc.mapping.IdColumnToPropertyMapping">
      <property name="columnName"><value>child_id</value></property>
      <property name="propertyName"><value>child_id</value></property>
      <property name="columnNameForVersion"><value>COALESCE(c.id, 0)</value></property>
    </bean>
  </list>
</property>
<property name="dataMappings">
  <list>
    <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>parent_first_name</value></property>
      <property name="propertyName"><value>parent_first_name</value></property>
    </bean>
    <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>child_first_name</value></property>
      <property name="propertyName"><value>child_first_name</value></property>
      <property name="propertyStoreString"><value>compress</value></property>
    </bean>
  </list>
</property>
<property name="versionMappings">
  <list>
    <bean class="org.compass.gps.device.jdbc.mapping.VersionColumnMapping">
      <property name="columnName"><value>parent_version</value></property>
    </bean>
    <bean class="org.compass.gps.device.jdbc.mapping.VersionColumnMapping">
      <property name="columnName"><value>child_version</value></property>
    </bean>
  </list>
</property>
</bean>

<!-- Compass-->
<bean id="compass" class="org.compass.spring.LocalCompassBean">
  <property name="mappingResolvers">
    <list>
      <bean class="org.compass.gps.device.jdbc.ResultSetResourceMappingResolver">
        <property name="mapping"><ref local="rsMapping" /></property>
        <property name="dataSource"><ref bean="dataSource" /></property>
      </bean>
    </list>
  </property>
  <property name="compassSettings">
    <props>
      <prop key="compass.engine.connection">target/testindex</prop>
      <!-- This is the default transaction handling (just explicitly setting it) -->
      <prop key="compass.transaction.factory">
        org.compass.core.transaction.LocalTransactionFactory
      </prop>
    </props>
  </property>
</bean>

<bean id="jdbcGpsDevice" class="org.compass.gps.device.jdbc.ResultSetJdbcGpsDevice">
  <property name="name"><value>jdbcDevice</value></property>
  <property name="dataSource"><ref bean="dataSource" /></property>
  <property name="mirrorDataChanges"><value>true</value></property>
  <property name="mappings">
    <list>
      <ref local="rsMapping" />
    </list>
  </property>
</bean>

<bean id="gps" class="org.compass.gps.impl.SingleCompassGps">
  <init-method="start" destroy-method="stop">
  <property name="compass"><ref bean="compass" /></property>
  <property name="gpsDevices">
    <list>
      <ref local="jdbcGpsDevice" />
    </list>
  </property>
  <property name="deleteIndexBeforeIndex"><value>true</value></property>
</bean>

</beans>

```

## 25.3. Table Mapping

A configuration sample of a the Table mapping given at the Jdbc Gps Device section is shown here in a Spring configuration file (taken from the unit tests):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="parentMapping" class="org.compass.gps.device.jdbc.mapping.TableToResourceMapping">
        <property name="alias"><value>parent</value></property>
        <property name="tableName"><value>PARENT</value></property>
        <property name="indexUnMappedColumns"><value>true</value></property>
        <property name="versionMappings">
            <list>
                <bean class="org.compass.gps.device.jdbc.mapping.VersionColumnMapping">
                    <property name="columnName"><value>version</value></property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="childMapping" class="org.compass.gps.device.jdbc.mapping.TableToResourceMapping">
        <property name="alias"><value>child</value></property>
        <property name="tableName"><value>CHILD</value></property>
        <property name="indexUnMappedColumns"><value>true</value></property>
        <property name="versionMappings">
            <list>
                <bean class="org.compass.gps.device.jdbc.mapping.VersionColumnMapping">
                    <property name="columnName"><value>version</value></property>
                </bean>
            </list>
        </property>
    </bean>

    <!-- Compass-->
    <bean id="compass" class="org.compass.spring.LocalCompassBean">
        <property name="mappingResolvers">
            <list>
                <bean class="org.compass.gps.device.jdbc.ResultSetResourceMappingResolver">
                    <property name="mapping"><ref local="parentMapping" /></property>
                    <property name="dataSource"><ref bean="dataSource" /></property>
                </bean>
                <bean class="org.compass.gps.device.jdbc.ResultSetResourceMappingResolver">
                    <property name="mapping"><ref local="childMapping" /></property>
                    <property name="dataSource"><ref bean="dataSource" /></property>
                </bean>
            </list>
        </property>
        <property name="compassSettings">
            <props>
                <prop key="compass.engine.connection">target/testindex</prop>
                <!-- This is the default transaction handling (just explicitly setting it) -->
                <prop key="compass.transaction.factory">
                    org.compass.core.transaction.LocalTransactionFactory
                </prop>
            </props>
        </property>
    </bean>

    <bean id="jdbcGpsDevice" class="org.compass.gps.device.jdbc.ResultSetJdbcGpsDevice">
        <property name="name"><value>jdbcDevice</value></property>
        <property name="dataSource"><ref bean="dataSource" /></property>
        <property name="mirrorDataChanges"><value>true</value></property>
        <property name="mappings">
            <list>
                <ref local="parentMapping" />
                <ref local="childMapping" />
            </list>
        </property>
        <property name="snapshotPersister">
            <bean class="org.compass.gps.device.jdbc.snapshot.FSJdbcSnapshotPersister">

```

```
        <property name="path"><value>target/testindex/snapshot</value></property>
    </bean>
</property>
</bean>

<bean id="gps" class="org.compass.gps.impl.SingleCompassGps"
        init-method="start" destroy-method="stop">
    <property name="compass"><ref bean="compass" /></property>
    <property name="gpsDevices">
        <list>
            <ref local="jdbcGpsDevice" />
        </list>
    </property>
    <property name="deleteIndexBeforeIndex"><value>true</value></property>
</bean>

</beans>
```

---

# Chapter 26. Spring AOP

## 26.1. Introduction

Compass provides a set of Spring AOP Advices which helps to mirror data changes done within a Spring powered application. For applications with a data source or a tool with no gps device that works with it (or it does not have mirroring capabilities - like iBatis), the mirror advices can make synchronizing changes made to the data source and Compass index simpler.

## 26.2. Advices

The AOP support comes with three advices: `CompassCreateAdvice`, `CompassSaveAdvice`, and `CompassDeleteAdvice`. They can create, save, or delete a data Object respectively. The advices are of type `AfterReturningAdvice`, and will persist the change to the index after the method proxied/adviced returns.

The data object that will be used to be created/saved/deleted can be one of the adviced method parameters (using the `parameterIndex` property), or it's return value (setting the `useReturnValue` to true).

## 26.3. Dao Sample

The following is an example using Spring AOP to proxy the dao layer. The Dao layer usually acts as an abstraction layer on top of the actual data source interaction code. It is one of the most common places where the Compass advices can be applied (the second is explained in the next section). The assumption here is that the Dao have create/save/delete methods.

```
<beans>
  ...

  <bean id="compass" class="org.compass.spring.LocalCompassBean">
    ... // configure a compass instance
  </bean>

  <!-- define the daos -->

  <bean id="userDao" class="eg.UserDaoImpl">
    ...
  </bean>

  <bean id="contactDao" class="eg.ContactDaoImpl">
    ...
  </bean>

  <!-- Definen the advisors -->

  <bean id="compassCreateAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="org.compass.spring.aop.CompassCreateAdvice">
        <property name="compass" ref="compass" />
      </bean>
    </property>
    <property name="pattern" value=".*create" />
  </bean>

  <bean id="compassSaveAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="org.compass.spring.aop.CompassSaveAdvice">
        <property name="compass" ref="compass" />
      </bean>
    </property>
    <property name="pattern" value=".*save" />
  </bean>
```

```

</bean>

<bean id="compassDeleteAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <bean class="org.compass.spring.aop.CompassDeleteAdvice">
      <property name="compass" ref="compass" />
    </bean>
  </property>
  <property name="pattern" value=".*delete" />
</bean>

<!-- Auto proxy the daos -->

<bean id="proxyCreator" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames"><value>userDao, contactDao</value></property>
  <property name="interceptorNames">
    <list>
      <value>compassCreateAdvisor</value>
      <value>compassSaveAdvisor</value>
      <value>compassDeleteAdvisor</value>
    </list>
  </property>
</bean>

...
</beans>

```

## 26.4. Transactional Service Sample

The following is an example using Spring AOP to proxy the transactional service layer. The service layer is already proxied by the `TransactionProxyFactoryBean`, and the Compass advices can be one of its `postInterceptors`. The assumption here is that the service layer have create/save/delete methods.

```

<beans>
  ...

  <bean id="compass" class="org.compass.spring.LocalCompassBean">
    ... // configure a compass instance
  </bean>

  <!-- Definen the advisors -->

  <bean id="compassCreateAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="org.compass.spring.aop.CompassCreateAdvice">
        <property name="compass" ref="compass" />
      </bean>
    </property>
    <property name="pattern" value=".*create" />
  </bean>

  <bean id="compassSaveAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="org.compass.spring.aop.CompassSaveAdvice">
        <property name="compass" ref="compass" />
      </bean>
    </property>
    <property name="pattern" value=".*save" />
  </bean>

  <bean id="compassDeleteAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <bean class="org.compass.spring.aop.CompassDeleteAdvice">
        <property name="compass" ref="compass" />
      </bean>
    </property>
    <property name="pattern" value=".*delete" />
  </bean>

  <!-- the transaciton proxy template -->

  <bean id="txProxyTemplate" abstract="true"

```

```
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
<property name="transactionManager"><ref bean="transactionManager"/></property>
<property name="transactionAttributes">
  <props>
    <prop key="create*">PROPAGATION_REQUIRED</prop>
    <prop key="save*">PROPAGATION_REQUIRED</prop>
    <prop key="delete*">PROPAGATION_REQUIRED</prop>
    <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
  </props>
</property>
</bean>

<bean id="userManager" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO"><ref bean="userDAO"/></property>
    </bean>
  </property>
  <property name="postInterceptors">
    <list>
      <ref bean="compassCreateAdvisor"/>
      <ref bean="compassSaveAdvisor"/>
      <ref bean="compassDeleteAdvisor"/>
    </list>
  </property>
</bean>

...
</beans>
```

---

# Chapter 27. Spring MVC Support

## 27.1. Introduction

Compass::Spring provides helper and support classes that build and integrate with Spring web MVC support. It has several base class controller helpers, as well as search and index controllers.

## 27.2. Support Classes

Two abstract command controllers are provided. The `AbstractCompassCommandController` is a general base class for Spring's MVC Command Controller that use `Compass`. The `AbstractCompassGpsCommandController` is a general base class for the Command Controller that use `CompassGps`.

## 27.3. Index Controller

`CompassIndexController` is a Spring Command Controller that can handle `index()` operations to perform on a `CompassGps`.

The controller command is `CompassIndexCommand`. The `CompassIndexController` command controller will perform the `index()` operation only if the `doIndex` parameter is set.

The controller has two views to be set. The `indexView` is the view that holds the screen which initiates the index operation, and the `indexResultsView`, which shows the results of the index operation.

The results of the index operation will be saved under the `indexResultsName`, which defaults to "indexResults". The results use the `CompassIndexResults` class.

## 27.4. Search Controller

`CompassSearchController` is a general Spring's MVC Controller that perform the search operation of `Compass`.

The Controller performs the search operation on the `Compass` instance using the query supplied by the `CompassSearchCommand`. The command holds the query that will be executed, as well as the page number (using the pagination feature).

If you wish to enable the pagination feature, you must set the `pageSize` property on the controller, as well as providing the page number property on the `CompassSearchCommand`.

The controller has two views to be set, the `searchView`, which is the view that holds the screen which the user will initiate the search operation, and the `searchResultsView`, which will show the results of the search operation (they can be the same page).

The results of the search operation will be saved under the `searchResultsName`, which defaults to "searchResults". The results use the `CompassSearchResults` class.

Note, that if using the `SpringSyncTransactionFactory`, the `transactionManager` must be set. Since when using the spring sync transaction setting, a spring managed transactions must be in progress already. The controller will start a transaction using the given transaction manager.

---

# Part V. Compass::Samples

The Samples section lists and explains all the different samples that come with Compass.

---

---

# Chapter 28. Library Sample

## 28.1. Introduction

Compass::Samples [library] is a basic example, that highlights the main features of Compass::Core. The application contains a small library domain model, containing `Author`, `Article` and `Book` Objects.

You can find most of Compass::Core features used within the library sample, such as OSEM and Common Metadata. It executes as a unit test, using [JUnit](#) and can be used to search a predefined set of data. Modify the `LibraryTests` class to add your own test data and experiment with how easy it is to work with Compass. Enjoy.

## 28.2. Running The Sample

Running the library sample, you will need to have [Apache Ant](#) installed and have `ANT_HOME/bin` on your path. The following table lists the available ant targets.

**Table 28.1.**

Target	Description
usage (also the default target)	Lists all the available targets.
test	Runs the tests defined in the <code>LibraryTests</code> , also compiles sample (see the compile target).
compile	Compiles the tests and the source code into the <code>build/classes</code> directory. Also executes the common meta data task to generate the <code>Library</code> class out of the <code>library.cmd.xml</code> file into the source directory.
search	Executes the <code>LibraryTests</code> main method, which pre-populates the index with data. You can interactively provide a search query to execute a search on the index.

---

# Chapter 29. Petclinic Sample

## 29.1. Introduction

Compass::Samples [petclinic] is the Spring petclinic sample powered by Compass. The main aim of the sample is to show how simple it is to add compass to an application, especially if one of the frameworks the application uses is one of the ones compass seamlessly integrates with.

Integrating compass into the petclinic sample, did not require any Java code to be written. Although several unit tests were added (good programming practice). Integration consisted of extending the Spring configuration files and writing a search and index jsp pages. The following sections will explain how integration was achieved.

The Compass petclinic sample shows how to integrate Compass with Spring and other frameworks. An important note, of course, is that Compass can be integrated with applications that do not use the Spring framework. Although Spring does make integration a bit simpler (and building applications much simpler).

## 29.2. Running The Sample

To run the petclinic sample, you will to install [Apache Ant](#) and have ANT\_HOME/bin on your path. The following table lists the available ant targets.

**Table 29.1.**

Target	Description
usage (also the default target)	Lists all the available targets.
clean	Clean all the output dirs.
build	Compile main Java sources and copy libraries.
docs	Create complete Javadoc documentation.
warfile	Build the web application archive.
setupDB	Initialize the database.
tests	Run the tests (a database does not have to be running).
all	Clean, build, docs, warfile, tests.

## 29.3. Data Model In Petclinic

Petclinic data model is based on POJOs (Plain Old Java Objects), including `Pet`, `Vet`, `Owner`, and `Visit` among others. The model also includes the base classes `Entity` (which holds the id of an entity), `NamedEntity` (holds a name as well), and `Person` (holds person information).

### 29.3.1. Common Meta-data (Optional)

As we explained in the Common Meta-data section, Common meta-data provides a global lookup mechanism for meta-data and alias definitions. It integrates with OSEM definitions and Gps::Jdbc mappings, externalising (and centralising) the actual semantic lookup keys that will be stored in the index. It also provides an Ant task to provides a constant Java class definitions for all the common meta-data definitions which can be used by Java application to lookup and store `Compass Resource`.

Defining a common meta-data definition is an optional step when integrating Compass. Though taking the time and creating one can provides valuable information and centralisation of the system (or systems) semantic definitions.

In the petclinic sample, the Common meta-data file is located in the `org.compass.sample.petclinic` package, and is called `petclinic.cmd.xml`. A fragment of it is shown here:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-meta-data PUBLIC
  "-//Compass/Compass Core Meta Data DTD 1.0//EN"
  "http://www.opensymphony.com/compass/dtd/compass-core-meta-data.dtd">

<compass-core-meta-data>

  <meta-data-group id="petclinic" displayName="Petclinic Meta Data">

    <description>Petclinic Meta Data</description>
    <uri>http://compass/sample/petclinic</uri>

    <alias id="vet" displayName="Vet">
      <description>Vet alias</description>
      <uri>http://compass/sample/petclinic/alias/vet</uri>
      <name>vet</name>
    </alias>

    <!-- ..... more alias definitions -->

    <meta-data id="birthdate" displayName="Birthdate">
      <description>The birthdate</description>
      <uri>http://compass/sample/petclinic/birthdate</uri>
      <name format="yyyy-MM-dd">birthdate</name>
    </meta-data>

    <!-- ..... more meta-data definitions -->

  </meta-data-group>

</compass-core-meta-data>
```

The above fragment of the common meta-data definitions, declares an alias called `vet` and meta-data called `birthdate`. The `birthdate` meta-data example shows one of the benefits of using common meta-data. The format of the date field is defined in the meta-data, instead of defining it in every mapping of `birthdate` meta-data (in OSEM for example).

### 29.3.2. Resource Mapping

One of the features of the search engine abstraction layer is the use of `Resource` and `Property`. As well as simple and minimal Resource Mapping definitions.

Although it is not directly used, the Jdbc implementation of the data access layer uses Search Engine API, based on `Resources` and resource mappings (the Jdbc device of `Compass::Gps` can automatically generate them).

### 29.3.3. OSEM

One of the main features of Compass is OSEM (Object / Search Engine Mapping), and it is heavily used in the petclinic sample. OSEM maps Java objects (domain model) to the underlying search engine using simple mapping definitions.

The petclinic sample uses most of the features provided by OSEM, among them are: `contract`, with mappings defined for the `Entity`, `NamedEntity`, and `Person` (all are "abstract" domain definitions), Cyclic references are defined (for example between pet and owner), and many more. The OSEM definitions can be found at the `petclinic.cpm.xml` file.

## 29.4. Data Access In Petclinic

Petclinic provides an abstraction layer on top of the actual implementation of the data access layer. The Petclinic can use Hibernate, Apache ORB, or JDBC for database access. Compass can seamlessly integrate with each of the mentioned layers.

The main concern with the data access layer (and Compass) is to synchronise each data model change made with Compass search engine index. Compass provides integration support for indexing the data using any of the actual implementation of the data access layer.

### 29.4.1. Hibernate

Compass::Gps comes with the Hibernate device. The device can index the data mapped by Hibernate, and mirror any data changes made by Hibernate to the search engine index. Since we are using Hibernate with Spring, the device used is the Spring Hibernate device.

The integration uses the OSEM definitions, working with Compass object level API to interact with the underlying search engine. The spring application context bean definitions for the `compass` (required by the Hibernate Gps device) instance is defined with OSEM definitions and spring based transaction support. The `applicationContext-hiberante.xml` in the test package, and the `applicationContext-hiberante.xml` in the WEB-INF directory define all the required definitions to work with hibernate and compass. Note, that only the mentioned configuration has to be created in order to integrate compass to the data access layer.

### 29.4.2. Apache OJB

Compass::Gps comes with the OJB device. The device can index the data mapped by OJB, and mirror any data changes made by OJB to the search engine index. Since we are using OJB with Spring, the device used is the Spring OJB device, which offers even simpler integration with OJB.

The integration uses the OSEM definitions, and works with Compass object level API to work with the search engine. The spring application context bean definitions for the `compass` (required by the OJB Gps device) instance is defined with OSEM definitions and spring based transaction support. The `applicationContext-ojb.xml` in the test package, and the `applicationContext-ojb.xml` in the WEB-INF directory define all the required definitions to work with OJB and compass. Note, that only the mentioned configuration has to be created in order to integrate compass to the data access layer.

### 29.4.3. JDBC

Compass::Gps comes with the JDBC device. The Jdbc device can connect to a database using `jdbc`, and based

on different mappings definitions, index it's content and mirror any data changes. When using the Jdbc device, the mapping is made on the `Resource` level (cannot use OSEM).

It is important to understand the different options for integrating Compass for a Jdbc (or a Jdbc helper framework like Spring or iBatis) data access implementation. If the system has no domain model, than `Resource` level API and mapping must be used. The Jdbc device can automate most of the actions needed to index and mirror the database. If the system has a domain model (such as the petclinic sample), two options are available: working on the `Resource` level and again using the Jdbc device, or using OSEM definitions, and plumb Compass calls to the data access API's (i.e. save the Vet in compass when the Vet is saved to the database). In the petclinic sample, the Jdbc device option was taken in order to demonstrate the Jdbc device usage. An API level solution should be simple, especially if the system has decent and centralized data access layer (which in our case it does).

The integration uses the Jdbc Gps Device mapping definitions and works with Compass object level API to work with the search engine. The spring application context bean definitions for the `compass` (required by the Jdbc Gps device) instance are defined with Jdbc mapping resolvers, and Local transactions. The `applicationContext-jdbc.xml` in the test package, and the `applicationContext-jdbc.xml` in the WEB-INF directory define all the required definitions to work with jdbc and compass. Note, that only the mentioned configuration has to be created in order to integrate compass to the data access layer.

The petclinic sample using the Jdbc Gps Device and defines several Jdbc mappings to the database. Some of the mappings use the more complex Result Set mappings (for mappings that require a join operation) and some use the simple Table mapping. The mapping definitions uses the common meta-data to lookup the actual meta-data values.

Note, that an important change made to the original petclinic sample was the addition the Version column. The version column is needed for automatic data mirroring (some databases, like Oracle, provides a "version column" by default).

The Resource mapping definition are automatically generated using mapping resolvers, and `compass` use them.

Note, that the Jdbc support currently only works with Hsql database (since the sql queries used in the Result Set mappings use hsql functions).

## 29.5. Web (MVC) in Petclinic

The petclinic sample uses Spring MVC framework for web support. `Compass::Spring` module comes with special support for the Spring MVC framework.

The only thing required when using the Compass and Spring mvc integration is writing the view layer for the search / index operations. These are the `index.jsp`, `search.jsp` and `searchResource.jsp` Jstl view based files.

The `index.jsp` is responsible for both initiating the index operation for `CompassGps` (and it's controlled devices), as well as displaying the results for the index operation.

The `search.jsp` and the `searchResource.jsp` are responsible for initiating the search operation as well as displaying the results. The difference between them is the `search.jsp` works with OSEM enabled petclinic (when using Hibernate or Apache OJB), and the `searchResource.jsp` works with resource mapping and resource level petclinic (when using Jdbc).

Note, that when using Jdbc, remember to change the `views.properties` file under the WEB-INF/classes directory and have both the `searchView.url` and the `searchResultsView.url` referring to `searchResource.jsp` view. And when using either Hibernate or OJB, change it to point to `search.jsp`.

---

# Part VI. Appendixes

---

---

# Appendix A. Configuration Settings

## A.1. Compass Configuration Settings

Compass's various settings have been logically grouped in the following section, with a short description of each setting. Note: the only mandatory setting is the index file location specified in `compass.engine.connection`.

Note, that configuring Compass is simpler when using a schema based configuration file. But in its core, all of Compass configuration is driven by the following settings. You can use only settings to configure Compass (either programatically or using the Compass configuration based on DTD).

### A.1.1. `compass.engine.connection`

Sets the Search engine index connection string.

**Table A.1.**

Connection	Description
<code>file:// prefix or no prefix</code>	The path to the file system based index path, using default file handling. This is a JVM level setting for all the file based prefixes.
<code>mmap:// prefix</code>	Uses Java 1.4 nio MMAP class. Considered slower than the general file system one, but might have memory benefits (according to the Lucene documentation). This is a JVM level setting for all the file based prefixes.
<code>ram:// prefix</code>	Creates a memory based index, follows the <code>Compass</code> life-cycle. Created when the <code>Compass</code> is created, and disposed when <code>Compass</code> is closed.
<code>jdbc:// prefix</code>	Holds the <code>Jdbc</code> url or <code>Jndi</code> (based on the <code>DataSourceProvider</code> configured). Allows storing the index within a database. This setting requires additional mandatory settings, please refer to the Search Engine <code>Jdbc</code> section. It is very IMPORTANT to read the Search Engine <code>Jdbc</code> section, especially in term of performance considerations.

### A.1.2. JNDI

Controls `Compass` registration through JNDI, using `Compass` JNDI lookups.

**Table A.2.**

Setting	Description
<code>compass.name</code>	The name that <code>Compass</code> will be registered under. Note that you can specify it at the XML configuration file with a <code>name</code> attribute at the

Setting	Description
	compass element.
compass.jndi.enable	Enables JNDI registration of compass under the given name. Default to false.
compass.jndi.class	JNDI initial context class, <code>Context.INITIAL_CONTEXT_FACTORY</code> .
compass.jndi.url	JNDI provider URL, <code>Context.PROVIDER_URL</code>
compass.jndi.*	prefix for arbitrary JNDI <code>InitialContext</code> properties

### A.1.3. Property

Controls `Compass` automatic properties, and property names.

**Table A.3.**

Setting	Description
compass.property.alias	The name of the "alias" property that Compass will use (a property that holds the alias property value of a resource). Defaults to <code>alias</code> (set it only if one of your mapped meta data is called alias).
compass.property.all	The name of the "all" property that Compass will use (a property that accumulates all the properties/meta-data). Defaults to <code>all</code> (set it only if one of your mapped meta data is called all). Note that it can be overridden in the mapping files.
compass.property.all.termVector (defaults to no)	The default setting for the term vector of the all property. Can be one of <code>no</code> , <code>yes</code> , <code>positions</code> , <code>offsets</code> , or <code>positions_offsets</code> .

### A.1.4. Transaction Level

Defines `Compass::Core` supported transaction and special transaction levels. The two most common transaction levels that `Compass::Core` supports are `read_committed` and `serializable` (`Compass::Core` uses a sophisticated mechanism for the `read_committed` level, which does operate as fast as the same transaction with a hint that it is read only). A special transaction level `batch_insert` is also supported, which specializes in handling batch indexing. You can set the transaction level using the `compass.transaction.isolation` setting. The following is a list of available `Compass` transaction levels:

**Table A.4.**

Transaction Level	Description
none	Not supported, upgraded to <code>read_committed</code> .
read_uncommitted	Not supported, upgraded to <code>read_committed</code> .
read_committed	The same read committed from data base systems. As fast for read

Transaction Level	Description
	only transactions.
repeatable_read	Not supported, upgraded to <code>serializable</code> .
serializable	The same as <code>serializable</code> from data base systems. Performance killer, basically results in transactions executed sequentially.
batch_insert	Specialized transaction level, mainly used for batch indexing. Note that it does not support queries, delete and save actions. Only the create operation is supported (If a resources with the same id and alias is in the index, you will have two after the create operation). Extremely fast for batch indexing, especially when tweaked with the matching settings in the Search Engine section.

Please read more about how `Compass::Core` implements it's transaction management in the Search Engine section.

### A.1.5. Transaction Strategy

When using the `Compass::Core` transaction API, you must specify a factory class for the `CompassTransaction` instances. This is done by setting the property `compass.transaction.factory`. The `CompassTransaction` API hides the underlying transaction mechanism, allowing `Compass::Core` code to run in a managed and non-managed environments. The two standard strategies are:

**Table A.5.**

Transaction Strategy	Description
<code>org.compass.core.transaction.LocalTransactionFactory</code>	Manages a local transaction which does not interact with other transaction mechanisms.
<code>org.compass.core.transaction.JTASyncTransactionFactory</code>	Uses the JTA synchronization support to synchronize with the JTA transaction (not the same as two phase commit, meaning that if the transaction fails, the other resources that participate in the transaction will not roll back). If there is no existing JTA transaction, a new one will be started.

An important configuration setting is the `compass.transaction.commitBeforeCompletion`. It is used when using transaction factories that uses synchronization (like JTA and Spring). If set to `true`, will commit the transaction in the `beforeCompletion` stage. It is very important to set it to `true` when using a jdbc based index storage, and set it to `false` otherwise. Defaults to `false`.

Although the J2EE specification does not provide a standard way to reference a JTA `TransactionManager`, to register with a transaction synchronization service, `Compass` provides several lookups which can be set with a `compass.transaction.managerLookup` setting (thanks hibernate!). The setting is not required since `Compass` will try to auto-detect the JTA environment.

The following table lists them all:

**Table A.6.**

Transaction Manager Lookup	Application Server
org.compass.core.transaction.manager.JBoss	JBoss
org.compass.core.transaction.manager.Weblogic	Weblogic
org.compass.core.transaction.manager.WebSphere	WebSphere
org.compass.core.transaction.manager.Orion	Orion
org.compass.core.transaction.manager.JOTM	JOTM
org.compass.core.transaction.manager.JOnAS	JOnAS
org.compass.core.transaction.manager.JRun4	JRun4
org.compass.core.transaction.manager.BEST	Borland ES

The JTA transaction mechanism will use the JNDI configuration to lookup the JTA `UserTransaction`. The transaction manager lookup provides the JNDI name, but if you wish to set it yourself, you can set the `compass.transaction.userTransactionName` setting. Also, the `UserTransaction` will be cached by default (fetched from JNDI on Compass startup), the caching can be controlled by `compass.transaction.cacheUserTransaction`.

### A.1.6. Property Accessor

Property accessors are used for reading and writing Class properties. Compass comes with two implementations, field and property. field is used for directly accessing the Class property, and property is used for accessing the class property using the property getters/setters. Compass allows for registration of custom `PropertyAccessor` implementations under a lookup name, as well as changing the default property accessor used (which is property).

The configuration uses Compass support for group properties, with the `compass.propertyAccessor` group prefix. The name the property accessor will be registered under is the group name. In order to set the default property accessor, the `default` group name should be used.

Custom implementations of `PropertyAccessor` can optionally implement the `CompassConfigurable` interface, which allows for additional settings to be injected into the implementations.

**Table A.7. Property Accessor Settings**

Setting	Description
<code>compass.propertyAccessor.[property accessor name].type</code>	The fully qualified class name of the property accessor.

### A.1.7. Converters

Compass uses converters to convert all the different OSEM mappings into `Resources`. Compass comes with a set of default converters that should be sufficient for most applications, but does allow the extendibility to define custom converters for all aspects related to marshaling Objects and Mappings (Compass internal mapping definitions) into a search engine.

Compass uses a registry of Converters. All Converters are registered under a registry name (converter lookup name). Compass registers all its default Converters under lookup names (which allows for changing the default converters settings), and allows for registration of custom Converters.

The following lists all the default Converters that comes with Compass. The lookup name is the lookup name the Converter will be registered under, the Converter class is Compass implementation of the `Converter`, and the Converter Type acts as shorthand string for the `Converter` implementation (can be used with the `compass.converter.[converter name].type` instead of the fully qualified class name). The default mapping converters are responsible for converting the meta-data mapping definitions.

**Table A.8. Default Compass Converters**

Java type	Lookup Name	Converter Class	Converter Type	Notes
java.lang.Boolean, boolean	boolean	org.compass.core.converter.simple.BooleanConveter	boolean	
java.lang.Byte, byte	byte	org.compass.core.converter.simple.ByteConveter	byte	
java.lang.Charecter, char	char	org.compass.core.converter.simple.CharConveter	char	
java.lang.Float, float	float	org.compass.core.converter.simple.FloatConveter	float	Format-table converter
java.lang.Double, double	double	org.compass.core.converter.simple.DoubleConveter	double	Format-table converter
java.lang.Short, short	short	org.compass.core.converter.simple.ShortConveter	short	Format-table converter
java.lang.Integer, int	int	org.compass.core.converter.simple.IntConveter	int	Format-table converter
java.lang.Long, long	long	org.compass.core.converter.simple.LongConveter	long	Format-table converter
java.lang.Date	date	org.compass.core.converter.simple.DateConveter	date	Format-table converter, defaults to yyyy-MM-dd-HH-mm-ss-S-a
java.lang.Calendar	calendar	org.compass.core.converter.simple.CalendarConveter	calendar	Format-table converter, defaults to yyyy-MM-dd-HH-mm-ss-S-a
java.lang.String	string	org.compass.core.converter.simple.StringConveter	string	
java.lang.StringBuffer	stringbuffer	org.compass.core.converter.simple.StringBufferConveter	stringbuffer	
java.math.BigDecimal	bigdecimal	org.compass.core.converter.simple.BigDecimalConveter	bigdecimal	

Java type	Lookup Name	Converter Class	Converter Type	Notes
java.math.BigInteger	biginteger	org.compass.core.converter.simple.BigIntegerConveter	biginteger	
java.net.URL	url	org.compass.core.converter.simple.URLConveter	url	Uses the URL#toString
java.io.File	file	org.compass.core.converter.extended.FileConveter	file	Uses the file name
java.io.InputStream	inputstream	org.compass.core.converter.extended.InputStreamConveter	inputstream	Stores the content of the <code>InputStream</code> without performing any other search related operations.
java.io.Reader	reader	org.compass.core.converter.extended.ReaderConverter	reader	
java.util.Locale	locale	org.compass.core.converter.extended.LocaleConveter	locale	
java.sql.Date	sqldate	org.compass.core.converter.extended.SqlDateConveter	sqldate	
java.sql.Time	sqltime	org.compass.core.converter.extended.SqlTimeConveter	sqltime	
java.sql.Timestamp	sqltimestamp	org.compass.core.converter.extended.SqlTimestampConveter	sqltimestamp	
byte[]	primitivebytearray	org.compass.core.converter.extended.PrimitiveByteArrayConverter	primitivebytearray	Stores the content of the byte array without performing any other search related operations.
Byte[]	objectbytearray	org.compass.core.converter.extended.ObjectByteArrayConverter	objectbytearray	Stores the content of the byte array without performing any other search related operations.

**Table A.9. Compass Mapping Converters**

Mapping type	Lookup Name	Converter Class	Notes
org.compass.core.mapping.osem.ClassMapping	classMapping	org.compass.core.converter.mapping.ClassMappingConverter	
org.compass.core.mapping.osem.ClassIdPropertyMapping	classIdPropertyMapping	org.compass.core.converter.mapping.ClassPropertyMappingConverter	
org.compass.core.mapping.osem.ClassPropertyMapping	classPropertyMapping	org.compass.core.converter.mapping.ClassPropertyMappingConverter	

Mapping type	Lookup Name	Converter Class	Notes
org.compass.core.mapping.osem.ComponentMapping	componentMapping	org.compass.core.converter.mapping.ComponentMappingConverter	
org.compass.core.mapping.osem.ReferenceMapping	referenceMapping	org.compass.core.converter.mapping.ReferenceMappingConverter	
org.compass.core.mapping.osem.CollectionMapping	collectionMapping	org.compass.core.converter.mapping.CollectionMappingConverter	
org.compass.core.mapping.osem.ArrayMapping	arrayMapping	org.compass.core.converter.mapping.ArrayMappingConverter	
org.compass.core.mapping.osem.ConstantMapping	constantMapping	org.compass.core.converter.mapping.ConstantMappingConverter	
org.compass.core.mapping.osem.ParentMapping	parentMapping	org.compass.core.converter.mapping.ParentMappingConverter	

Defining a new converter can be done using Compass support for group settings. `compass.converter` is the prefix for the group. In order to define new converter that will be registered under the "converter name" lookup, the `compass.converter.[converter name]` setting prefix should be used. The following lists all the settings that can apply to a converter definition.

**Table A.10. Converter Settings**

Setting	Description
<code>compass.converter.[converter name].type</code>	The type of the <code>org.compass.converter.Converter</code> implementation. Should either be the fully qualified class name, or the Converter Type (shorthand version for compass internal converter classes, defined in the previous table).
<code>compass.converter.[converter name].type</code>	The type of the <code>org.compass.converter.Converter</code> implementation. Should either be the fully qualified class name, or the Converter Type (shorthand version for compass internal converter classes, defined in the previous table).
<code>compass.converter.[converter name].format</code>	Applies to format-able converters. The format that will be used to format the data converted (see Java <code>java.text.DecimalFormat</code> and <code>java.text.SimpleDateFormat</code> ).
<code>compass.converter.[converter name].format.locale</code>	The <code>Locale</code> to be used when formatting.
<code>compass.converter.[converter name].format.minPoolSize</code>	Compass pools the formatters for greater performance. The value of the minimum pool size. Defaults to 4.
<code>compass.converter.[converter name].format.maxPoolSize</code>	Compass pools the formatters for greater performance. The value of the maximum pool size. Defaults to 20.

Note, that any other setting can be defined after the `compass.converter.[converter name]`. If the Converter implements the `org.compass.core.config.CompassConfigurable`, it will be injected with the settings for the converter. The converter will get all the settings, with settings names without the `compass.converter.[converter name]` prefix.

For example, defining a new Date converter with a specific format can be done by setting two settings:  
`compass.converter.mydate.type=date` (same as  
`compass.converter.mydate.type=org.compass.core.converter.basic.DateConverter`), and  
`compass.converter.mydate.format=yyyy-HH-dd`. The converter will be registered under the "mydate"  
 converter lookup name. It can then be used as a lookup name in the OSEM definitions.

In order to change the default converters, simply define a setting with the [converter name] of the default  
 converter that comes with compass. For example, in order to override the format of all the dates in the system  
 to "yyyy-HH-dd", simple set: `compass.converter.date.format=yyyy-HH-dd`.

### A.1.8. Search Engine

Controls the different settings for the search engine.

**Table A.11. Search Engine Settings**

Setting	Description
<code>compass.engine.connection</code>	The index engine file system location.
<code>compass.engine.defaultsearch</code>	When searching using a query string, the default property/meta-data that compass will use for non prefixed strings. Defaults to <code>compass.property.all</code> value.
<code>compass.engine.all.analyzer</code>	The name of the analyzer to use for the all property (see the next section about Search Engine Analyzers).
<code>compass.transaction.clearCacheOnCommit</code>	Should the cache be cleared on commit. Note, that setting it to <code>false</code> might mean that the transaction isolation level will not function properly (for example, with <code>read_committed</code> , it will mean that data that is committed will take time to be reflected in the index). Defaults to <code>true</code> .
<code>compass.transaction.lockDir</code>	The directory where the search engine will maintain it's locking file mechanism for inter and outer process transaction synchronization. Defaults to the <code>java.io.tmpdir</code> Java system property. This is a JVM level property.
<code>compass.transaction.lockTimeout</code>	The amount of time a transaction will wait in order to obtain it's specific lock (in seconds). Defaults to 10 seconds.
<code>compass.transaction.commitTimeout</code>	The amount of time a transaction will wait in order to commit it's data. Defaults to 10 seconds.
<code>compass.transaction.lockPollInterval</code>	The interval that the transaction will check to see if it can obtain the lock (in milliseconds). Defaults to 100 milliseconds. This is a JVM level property.
<code>compass.engine.optimizer.type</code>	The fully qualified class name of the search engine optimizer that will be used. Defaults to <code>org.compass.core.lucene.engine.optimizer.AdaptiveOptimizer</code> . Please see the following section for a list of optimizers.
<code>compass.engine.optimizer.schedule</code>	Determines if the optimizer will be scheduled or not ( <code>true</code> or <code>false</code> ), defaults to <code>true</code> . If it is scheduled, it will run each period of time and check if the index need optimization, and if it does, it will

Setting	Description
	optimize it.
compass.engine.optimizer.schedule.period	The period that the optimizer will check if the index need optimization, and if it does, optimize it (in seconds, can be a float number). Defaults to 10 seconds. The setting applies if the optimizer is scheduled.
compass.engine.optimizer.schedule.daemon	Sets the optimizer thread to be a daemon ( <code>true</code> ) or not ( <code>false</code> ). Defaults to <code>true</code> . The setting applies if the optimizer is scheduled.
compass.engine.optimizer.schedule.fixedRate	Determines if the schedule will run in a fixed rate or not. If it is set to <code>false</code> each execution is scheduled relative to the actual execution of the previous execution. If it is set to <code>true</code> each execution is scheduled relative to the execution time of the initial execution.
compass.engine.optimizer.adaptive.mergeFactor	For the adaptive optimizer, determines how often the optimizer will optimize the index. With small values, the faster the searches will be, but the more often that the index will be optimized. Larger values will result in slower searches, and less optimizations.
compass.engine.optimizer.aggressive.mergeFactor	For the aggressive optimizer, determines how often the optimizer will optimize the index. With small values, the faster the searches will be, but the more often that the index will be optimized. Larger values will result in slower searches, and less optimizations.
compass.engine.mergeFactor	Applies only for <code>batch_insert</code> transaction. With smaller values, less RAM is used, but indexing is slower. With larger values, more RAM is used, and the indexing speed is faster. Defaults to 10.
compass.engine.maxBufferedDocs	Applies only for <code>batch_insert</code> transaction. Determines the minimal number of resources required before the buffered in-memory resources will be flushed to disk. Large values give faster indexing. At the same time, <code>compass.engine.mergeFactor</code> limits the number of files open.
compass.engine.maxFieldLength	The number of terms that will be indexed for a single property in a resource. This limits the amount of memory required for indexing, so that collections with very large resources will not crash the indexing process by running out of memory. Note, that this effectively truncates large resources, excluding from the index terms that occur further in the resource. Defaults to 10,000 terms.
compass.engine.useCompoundFile	Turn on ( <code>true</code> ) or off ( <code>false</code> ) the use of compound files. If used lowers the number of files open, but have very small performance overhead. Defaults to <code>true</code> . Note, when compass starts up, it will validate that the current index structure maps the configured setting, and if it is not, it will automatically try and convert it to the correct structure.
compass.engine.cacheIntervalInvalidation	Sets how often (in milliseconds) the index manager will check if the index cache needs to be invalidated. Defaults to 5000 milliseconds. Setting it to 0 means that the cache will check if it needs to be invalidated all the time. Setting it to -1 means that the cache will not check the index for invalidation, it is perfectly fine if

Setting	Description
	a single instance is working with the index, since the cache is automatically invalidated upon a dirty operation.
<code>compass.engine.indexManagerScheduleInterval</code>	The index manager schedule interval (in seconds) where different actions related to index manager will happen (such as global cache interval invalidation checks - see <code>SearchEngineIndexManager#notifyAllToClearCache</code> and <code>SearchEngineIndexManager#checkAndClearIfNotifiedAllToClearCache</code> ). Defaults to 60 seconds.
<code>compass.engine.waitForCacheInvalidationOnIndexOperation</code>	<b>Default: false</b> If set to true, will cause the index manager operation (including replace index) to wait for all other compass instances to invalidate their cache. The time to wait will be the <code>indexManagerScheduledInterval</code> configuration setting.

The following section lists the different optimizers that are available with `Compass::Core`. Note that all the optimizers can be scheduled or not.

**Table A.12.**

Optimizer	Description
<code>org.compass.core.lucene.engine.optimizer.AdaptiveOptimizer</code>	When the number of segments exceeds that specified <code>mergeFactor</code> , the segments will be merged from the last segment, until a segment with a higher resource count will be encountered.
<code>org.compass.core.lucene.engine.optimizer.AggressiveOptimizer</code>	When the number of segments exceeds that specified <code>mergeFactor</code> , all the segments are merged into a single segment.
<code>org.compass.core.lucene.engine.optimizer.NullOptimizer</code>	Does no optimization, starts no threads.

### A.1.9. Search Engine Jdbc

Compass allows storing the index in a database using Jdbc. When using Jdbc storage, additional settings are mandatory except for the connection setting. The value after the `Jdbc://` prefix in the `compass.engine.connection` setting can be the Jdbc url connection or the Jndi name of the `DataSource`, depending on the configured `DataSourceProvider`.

It is important also to read the Jdbc Directory Appendix. Two sections that should be read are the supported dialects, and the performance considerations (especially the compound structure).

The following is a list of all the Jdbc settings:

**Table A.13. Search Engine Jdbc Settings**

Setting	Description
<code>compass.engine.store.jdbc.dialect</code>	Optional. The fully qualified class name of the dialect (the database type) that the index will be stored at. Please refer to Lucene Jdbc Directory appendix for a list of the currently supported dialects. If

Setting	Description
	not set, Compass will try to auto-detect it based on the Database meta-data.
compass.engine.store.jdbc.disableSchemaOperations	Optional. If set to <code>true</code> , no database schema level operations will be performed (drop and create tables). When deleting the data in the index, the content will be deleted, but the table will not be dropped. Default to <code>false</code> .
compass.engine.store.jdbc.managed	Optional (defaults to <code>false</code> ). If the connection is managed or not. Basically, if set to <code>false</code> , compass will commit and rollback the transaction. If set to <code>true</code> , compass will not perform it. Defaults to <code>false</code> . Should be set to <code>true</code> if using external transaction managers (like JTA or Spring <code>PlatformTransactionManager</code> ), and <code>false</code> if using compass <code>LocalTransactionFactory</code> . Note as well, that if using external transaction managers, the <code>compass.transaction.commitBeforeCompletion</code> should be set to <code>true</code> . If the connection is not managed (set to <code>false</code> ), the created <code>DataSource</code> will be wrapped with <code>Compass Jdbc directory TransactionAwareDataSourceProxy</code> . Please refer to <code>Lucene Jdbc Directory</code> appendix for more information.
compass.engine.store.jdbc.connection.provider.class	The fully qualified name of the <code>DataSourceProvider</code> . The <code>DataSourceProvider</code> is responsible for getting/creating the <code>Jdbc DataSource</code> that will be used. Defaults to <code>org.compass.core.lucene.engine.store.jdbc.DriverManagerDataSourceProvider</code> (Poor performance). Please refer to next section for a list of the available providers.
compass.engine.store.jdbc.useCommitLocks	Optional (defaults to <code>false</code> ). Determines if the index will use Lucene commit locks. Setting it to <code>true</code> makes sense only if the system will work in <code>autoCommit</code> mode (which is not recommended anyhow).
compass.engine.store.jdbc.deleteMarkDeletedDelta	Optional (defaults to an hour). Some of the entries in the database are marked as deleted, and not actually gets to be deleted from the database. The setting controls the delta time of when they should be deleted. They will be deleted if they were marked for deleted "delta" time ago (base on database time, if possible by dialect).
compass.engine.store.jdbc.lockType	Optional (defaults to <code>PhantomReadLock</code> ). The fully qualified name of the <code>Lock</code> implementation that will be used.
compass.engine.store.jdbc.ddl.name.name	Optional (defaults to <code>name_</code> ). The name of the name column.
compass.engine.store.jdbc.ddl.name.size	Optional (defaults to 50). The size (charecters) of the name column.
compass.engine.store.jdbc.ddl.value.name	Optional (defaults to <code>value_</code> ). The name of the value column.
compass.engine.store.jdbc.ddl.value.size	Optional (defaults to 500 * 1000 K). The size (in K) of the value column. Only applies to databases that require it.
compass.engine.store.jdbc.	Optional (defaults to <code>size_</code> ). The name of the size column.

Setting	Description
ddl.size.name	
compass.engine.store.jdbc.ddl.lastModified.name	Optional (defaults to <code>lf_</code> ). The name of the last modified column.
compass.engine.store.jdbc.ddl.deleted.name	Optional (defaults to <code>deleted_</code> ). The name of the deleted column.

### A.1.9.1. Data Source Providers

Compass comes with several built in `DataSourceProvider`s. They are all located at the `org.compass.core.lucene.engine.store.jdbc` package. The following table lists them:

**Table A.14. Search Engine Data Source Providers**

Data Source Provider Class	Description
<code>DriverManagerDataSourceProvider</code>	The default data source provider. Creates a simple <code>DataSource</code> that returns a new <code>Connection</code> for each request. Performs very poorly, and should not be used.
<code>DbcpDataSourceProvider</code>	Uses Jakarta Commons DBCP Connection pool. Compass provides several additional configurations settings to configure DBCP, please refer to <code>LuceneEnvironment#DataSourceProvider#Dbcp</code> javadoc.
<code>C3P0DataSourceProvider</code>	Uses C3P0 Connection pool. Configuring additional properties for the C3P0 connection pool uses C3p0 internal support for a <code>c3p0.properties</code> that should reside as a top-level resource in the same CLASSPATH / classloader that loads c3p0's jar file.
<code>JndiDataSourceProvider</code>	Gets a <code>DataSource</code> from JNDI. The JNDI name is the value after the <code>jdbc://</code> prefix in Compass connection setting.
<code>ExternalDataSourceProvider</code>	A data source provider that can use an externally configured data source. In order to set the external <code>DataSource</code> to be used, the <code>ExternalDataSourceProvider#setDataSource(DataSource)</code> static method needs to be called before the <code>Compass</code> instance is created.

The `DriverManagerDataSourceProvider`, `DbcpDataSourceProvider`, and `C3P0DataSourceProvider` use the value after the `jdbc://` prefix in Compass connection setting as the `Jdbc` connection url. They also require the following settings to be set:

**Table A.15. Internal Data Source Providers Settings**

Setting	Description
compass.engine.store.jdbc.connection.driverClass	The <code>Jdbc</code> driver class.
compass.engine.store.jdbc.	The <code>Jdbc</code> connection user name.

Setting	Description
connection.username	
compass.engine.store.jdbc.connection.password	The Jdbc connection password.

### A.1.9.2. File Entry Handlers

Configuring the Jdbc store with Compass also allows defining `FileEntryHandler` settings for different file entries in the database. `FileEntryHandlers` are explained in the Lucene Jdbc Directory appendix (and require some Lucene knowledge). The Lucene Jdbc Directory implementation already comes with sensible defaults, but they can be changed using Compass configuration.

One of the things that come free with Compass it automatically using the more performant `FetchPerTransactionJdbcIndexInput` if possible (based on the dialect). Special care need to be taken when using the mentioned index input, and it is done automatically by Compass.

Setting file entry handlers is done using the following setting prefix: `compass.engine.store.jdbc.fe.[name]`. The name can be either `__default__` which is used for all unmapped files, it can be the full name of the file stored, or the suffix of the file (the last 3 charecters). Some of the currently supported settings are:

**Table A.16. File Entry Handler Settings**

Setting	Description
compass.engine.store.jdbc.fe.[name].type	The fully qualified class name of the file entry handler.
compass.engine.store.jdbc.fe.[name].indexInput.type	The fully qualified class name of the <code>IndexInput</code> implementation.
compass.engine.store.jdbc.fe.[name].indexOutput.type	The fully qualified class name of the <code>IndexInput</code> implementation.
compass.engine.store.jdbc.fe.[name].indexInput.bufferSize	The RAM buffer size of the index input. Note, it applies only to some of the <code>IndexInput</code> implementations.
compass.engine.store.jdbc.fe.[name].indexOutput.bufferSize	The RAM buffer size of the index output. Note, it applies only to some of the <code>IndexOutput</code> implementations.
compass.engine.store.jdbc.fe.[name].indexOutput.threshold	The threshold value (in bytes) after which data will be temporarily written to a file (and them dumped into the database). Applies when using <code>RAMAndFileJdbcIndexOutput</code> (which is the default one). Defaults to $16 * 1024$ bytes.

### A.1.10. Search Engine Analyzers

With Compass, multiple Analyzers can be defined (each under a different analyzer name) and than referenced in the configuration and mapping definitions. Compass defines two internal analyzers names called: `default` and `search`. The `default` analyzer is the one used when no other analyzer can be found, it defaults to the standard analyzer with English stop words. The `search` is the analyzer used to analyze search queries, and if not set, defaults to the `default` analyzer (Note that the search analyzer can also be set using the `CompassQuery`

API). Changing the settings for the default analyzer can be done using the `compass.engine.analyzer.default.*` settings (as explained in the next table). Setting the search analyzer (so it will differ from the default analyzer) can be done using the `compass.engine.analyzer.search.*` settings. Also, you can set a list of filter to be applied to the given analyzers, please see the next section of how to configure analyzer filters, especially the synonym one.

**Table A.17. Search Engine Analyzer Settings**

Setting	Description
<code>compass.engine.analyzer.[analyzer name].type</code>	The type of the search engine analyzer, please see the available analyzers types later in the section.
<code>compass.engine.analyzer.[analyzer name].filters</code>	A comma separated list of <code>LuceneAnalyzerTokenFilterProviders</code> registered under compass, to be applied for the given analyzer. For example, adding a synonym analyzer, you should register a synonym <code>LuceneAnalyzerTokenFilterProvider</code> under your own choice for filter name, and add it to the list of filters here.
<code>compass.engine.analyzer.[analyzer name].stopwords</code>	A comma separated list of stop words to use with the chosen analyzer. If the string starts with <code>+</code> , the list of stop-words will be added to the default set of stop words defined for the analyzer. Note, that not all the analyzers type support this feature.
<code>compass.engine.analyzer.[analyzer name].factory</code>	If the <code>compass.engine.analyzer.[analyzer name].type</code> setting is not enough to configure your analyzer, use it to define the fully qualified class name of your analyzer factory which implements <code>LuceneAnalyzerFactory</code> class.

Compass comes with core analyzers (Which are part of the `lucene-core` jar). They are: `standard`, `simple`, `whitespace`, and `stop`. See the Analyzers Section.

Compass also allows simple configuration of the `snowball` analyzer type (which comes with the `lucene-snowball` jar). An additional setting that must be set when using the `snowball` analyzer, is the `compass.engine.analyzer.[analyzer name].name` setting. The settings can have the following values: `Danish`, `Dutch`, `English`, `Finnish`, `French`, `German`, `German2`, `Italian`, `Kp`, `Lovins`, `Norwegian`, `Porter`, `Portuguese`, `Russian`, `Spanish`, and `Swedish`.

Another set of analyzer types comes with the `lucene-analyzers` jar. They are: `brazilian`, `cyj`, `chinese`, `czech`, `german`, `greek`, `french`, `dutch`, and `russian`.

### A.1.11. Search Engine Analyzer Filters

You can specify a set of analyzer filters that can then be applied to all the different analyzers configured. It uses the group settings, so setting the analyzer filter need to be prefixed with `compass.engine.analyzerfilter`, and the value after it is the analyzer filter name, and then the setting for the analyzer filter.

Filters are provided for simpler support for additional filtering (or enrichment) of analyzed streams, without the hassle of creating your own analyzer. Also, filters, can be shared across different analyzers, potentially having different analyzer types.

**Table A.18.**

Setting	Description
compass.engine.analyzerfilter.[analyzer filter name].type	The type of the search engine analyzer filter provider, must implement the <code>org.compass.core.lucene.engine.analyzer.LuceneAnalyzerTokenFilterProvider</code> interface. Can also be the value <code>synonym</code> , which will automatically map to the <code>org.compass.core.lucene.engine.analyzer.synonym.SynonymAnalyzerTokenFilterProvider</code> class.
compass.engine.analyzerfilter.[analyzer filter name].lookup	Only applies for synonym filters. The class that implements the <code>org.compass.core.lucene.engine.analyzer.synonym.SynonymLookupProvider</code> interface for providing synonyms for a given term.

### A.1.12. Search Engine Highlighters

With Compass, multiple Highlighters can be defined (each under a different highlighter name) and then referenced when using `CompassHighlighter`. Within Compass, an internal `default` highlighter is defined, and can be configured when using `default` as the highlighter name.

**Table A.19.**

Setting	Description
compass.engine.highlighter.[highlighter name].factory	Low level. Optional (defaults to <code>DefaultLuceneHighlighterFactory</code> ). The fully qualified name of the class that creates highlighters settings. Must implement the <code>LuceneHighlighterFactory</code> interface.
compass.engine.highlighter.[highlighter name].textTokenizer	Optional (default to <code>auto</code> ). Defines how a text will be tokenized to be highlighted. Can be <code>analyzer</code> (use an analyzer to tokenize the text), <code>term_vector</code> (use the term vector info stored in the index), or <code>auto</code> (will first try <code>term_vector</code> , and if no info is stored, will try to use <code>analyzer</code> ).
compass.engine.highlighter.[highlighter name].rewriteQuery	Low level. Optional (defaults to <code>true</code> ). If the query used to highlight the text will be rewritten or not.
compass.engine.highlighter.[highlighter name].computeIdf	Low level. Optional (set according to the formatter used).
compass.engine.highlighter.[highlighter name].maxNumFragments	Optional (default to 3). Sets the maximum number of fragments that will be returned.
compass.engine.highlighter.[highlighter name].separator	Optional (defaults to <code>...</code> ). Sets the separator string between fragments if using the combined fragments highlight option.
compass.engine.highlighter.[highlighter name].maxBytesToAnalyze	Optional (defaults to <code>50*1024</code> ). Sets the maximum bytes of text to analyze.
compass.engine.highlighter.[highlighter name].fragmenter	Optional (default to <code>simple</code> ). The type of the fragmenter that will

Setting	Description
name].fragmenter.type	be used, can be <code>simple</code> , <code>null</code> , or the fully qualified class name of the <code>fragmenter</code> (implements the <code>org.apache.lucene.search.highlight.Fragmenter</code> ).
compass.engine.highlighter.[highlighter name].fragmenter.simple.size	Optional (defaults to 100). Sets the size (in bytes) of the fragments for the <code>simple</code> fragmenter.
compass.engine.highlighter.[highlighter name].encoder.type	Optional (default to <code>default</code> ). The type of the encoder that will be used to encode fragmented text. Can be <code>default</code> (does nothing), <code>html</code> (escapes html tags), or the fully qualified class name of the <code>encoder</code> (implements <code>org.apache.lucene.search.highlight.Encoder</code> ).
compass.engine.highlighter.[highlighter name].formatter.type	Optional (default to <code>simple</code> ). The type of the formatter that will be used to highlight the text. Can be <code>simple</code> (simply wraps the highlighted text with <code>pre</code> and <code>post</code> strings), <code>htmlSpanGradient</code> (wraps the highlighted text with an html span tag with an optional background and foreground gradient colors), or the fully qualified class name of the <code>formatter</code> (implements <code>org.apache.lucene.search.highlight.Formatter</code> ).
compass.engine.highlighter.[highlighter name].formatter.simple.pre	Optional (default to <code>&lt;b&gt;</code> ). In case the highlighter uses the <code>simple</code> formatter, controls the text that is appended before the highlighted text.
compass.engine.highlighter.[highlighter name].formatter.simple.post	Optional (default to <code>&lt;/b&gt;</code> ). In case the highlighter uses the <code>simple</code> formatter, controls the text that is appended after the highlighted text.
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.maxScore	In case the highlighter uses the <code>htmlSpanGradient</code> formatter, the score that above it is displayed as max color.
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.minForegroundColor	Optional (if not set, foreground will not be set on the span tag). In case the highlighter uses the <code>htmlSpanGradient</code> formatter, the hex color used for representing IDF scores of zero eg <code>#FFFFFF</code> (white).
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.maxForegroundColor	Optional (if not set, foreground will not be set on the span tag). In case the highlighter uses the <code>htmlSpanGradient</code> formatter, the largest hex color used for representing IDF scores eg <code>#000000</code> (black).
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.minBackgroundColor	Optional (if not set, background will not be set on the span tag). In case the highlighter uses the <code>htmlSpanGradient</code> formatter, the hex color used for representing IDF scores of zero eg <code>#FFFFFF</code> (white).
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.maxBackgroundColor	Optional (if not set, background will not be set on the span tag). In case the highlighter uses the <code>htmlSpanGradient</code> formatter, The largest hex color used for representing IDF scores eg <code>#000000</code> (black).

### A.1.13. Other Settings

Several other settings that control compass.

**Table A.20.**

<b>Setting</b>	<b>Description</b>
compass.managedId.index	Can be either <code>un_tokenized</code> or <code>no</code> (defaults to <code>no</code> ). It is the index setting that will be used when creating an internal managed id for a class property mapping (if it is not a property id, if it is, it will always be <code>un_tokenized</code> ).

---

# Appendix B. Lucene Jdbc Directory

## B.1. Overview

A Jdbc based implementation of Lucene `Directory` allowing the storage of a Lucene index within a database. Enables existing or new Lucene based application to store the Lucene index in a database with no or minimal change to typical Lucene code fragments.

The `JdbcDirectory` is highly configurable, using the optional `JdbcDirectorySettings`. All the settings are described in the javadoc, and most of them will be made clear during the next sections.

There are several options to instantiate a Jdbc directory, they are:

**Table B.1. Jdbc Directory Constructors**

Parameters	Description
<code>DataSource</code> , <code>Dialect</code> , <code>tableName</code>	Creates a new <code>JdbcDirectory</code> using the given data source and dialect. <code>JdbcTable</code> and <code>JdbcDirectorySettings</code> are created based on default values.
<code>DataSource</code> , <code>Dialect</code> , <code>JdbcDirectorySettings</code> , <code>tableName</code>	Creates a new <code>JdbcDirectory</code> using the given data source, dialect, and <code>JdbcDirectorySettings</code> . The <code>JdbcTable</code> is created internally.
<code>DataSource</code> , <code>JdbcTable</code>	Creates a new <code>JdbcDirectory</code> using the given dialect, and <code>JdbcTable</code> . Creating a new <code>JdbcTable</code> requires a <code>Dialect</code> and <code>JdbcDirectorySettings</code> .

The Jdbc directory works against a single table (where the table name must be provided when the directory is created). The table schema is described in the following table:

**Table B.2. Jdbc Directory Table Schema**

Column Name	Column Type	Default Column Name	Description
Name	VARCHAR	name_	The file entry name. Similar to a file name within a file system directory. The column size is configurable and defaults to 50.
Value	BLOB	value_	A binary column where the content of the file is stored. Based on Jdbc <code>Blob</code> type. Can have a configurable size where appropriate for the database type.
Size	NUMBER	size_	The size of the current saved data in the Value column. Similar to the size of a file in a file system.
Last Modified	TIMESTAMP	lf_	The time that file was last modified. Similar to the last modified time of a file within a file system.

Column Name	Column Type	Default Column Name	Description
Deleted	BIT	deleted_	If the file is deleted or not. Only used for some of the file types based on the Jdbc directory. More is explained in later sections.

The Jdbc directory provides the following operations on top of the ones forced by the `Directory` interface:

**Table B.3. Extended Jdbc Directory Operations**

Operation Name	Description
create	Creates the database table (with the above mentioned schema). The create operation drops the table first.
delete	Drops the table from the database.
deleteContent	Deletes all the rows from the table in the database.
tableExists	Returns if the table exists or not. Only supported on some of the databases.
deleteMarkDeleted	Deletes all the file entries that are marked to be deleted, and they were marked, and they were marked "delta" time ago (base on database time, if possible by dialect). The delta is taken from the <code>JdbcDirectorySettings</code> , or provided as a parameter to the <code>deleteMarkDeleted</code> operation.

The Jdbc directory requires a `Dialect` implementation that is specific to the database used with it. The following is a table listing the current dialects supported with the Jdbc directory:

**Table B.4. Jdbc Directory SQL Dialects**

Dialect	RDBMS	Blob Locator Support*
<code>org.apache.lucene.store.jdbc.dialect.Oracle</code>	Oracle	Oracle Jdbc Driver - Yes
<code>org.apache.lucene.store.jdbc.dialect.SQLServer</code>	Microsoft SQL Server	jTds 1.2 - No. Microsoft Jdbc Driver - Unknown
<code>org.apache.lucene.store.jdbc.dialect.MySQL</code>	MySQL	MySQL Connector J 3.1 - Yes with <code>emulateLocators=true</code> in connection string.
<code>org.apache.lucene.store.jdbc.dialect.MySQLInnoDB</code>	MySQL with InnoDB.	See MySQL
<code>org.apache.lucene.store.jdbc.dialect.MySQLMyISAM</code>	MySQL with MyISAM	See MySQL
<code>org.apache.lucene.store.jdbc.dialect.Postgres</code>	PostgreSQL	Postgres Jdbc Driver - Yes.
<code>org.apache.lucene.store.jdbc.dialect.Sybase</code>	Sybase/ Sybase Anywhere	Unknown.
<code>org.apache.lucene.store.jdbc.dialect.Interbase</code>	Interbase	Unknown.
<code>org.apache.lucene.store.jdbc.dialect.Firebird</code>	Firebird	Unknown.

Dialect	RDBMS	Blob Locator Support*
<code>org.apache.lucene.store.jdbc.dialect.DB2Dialect</code>	DB2 / DB2 AS400 / DB2 OS390	Unknown.
<code>org.apache.lucene.store.jdbc.dialect.DerbyDialect</code>	Derby	Derby Jdbc Driver- Unknown.
<code>org.apache.lucene.store.jdbc.dialect.HSQLDialect</code>	HypersonicSQL	HSQL Jdbc Driver - No.

\* A Blob locator is a pointer to the actual data, which allows fetching only portions of the Blob at a time. Databases (or Jdbc drivers) that do not use locators usually fetch all the Blob data for each query (which makes using them impractical for large indexes). Note, the support documented here does not cover all the possible Jdbc drivers, please refer to your Jdbc driver documentation for more information.

## B.2. Performance Notes

Minor performance improvements can be gained if `JdbcTable` is cached and used to create different `JdbcDirectory` instances.

It is best to use a pooled data source (like Jakarta Commons DBCP), so `Connections` won't get created every time, but be pooled.

Most of the time, when working with Jdbc directory, it is best to work in a non compound index format. Since with databases there is no problem of too many files open, it won't be an issue. The package comes with a set of utilities to compound or uncompound an index, located in the `org.apache.lucene.index.LuceneUtils` class, just in case you already have an index and it is in the wrong structure.

When indexing data, a possible performance improvement can be to index the data into the file system or memory, and then copy over the contents of the index to the database. `org.apache.lucene.index.LuceneUtils` comes with a utility to copy one directory to the other, and changing the compound state of the index while copying.

## B.3. Transaction Management

`JdbcDirectory` performs no transaction management. All database related operations WITHIN IT work in the following manner:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
// perform any database related operation using the connection
DataSourceUtils.releaseConnection(conn);
```

As you can see, no `commit` or `rollback` are called on the connection, allowing for any type of transaction management done outside of the actual `JdbcDirectory` related operations. Also, the fact that we are using the Jdbc `DataSource`, allows for plug able transaction management support (usually based on `DataSource` delegate and `Connection` proxy). `DataSourceUtils` is a utility class that comes with the Jdbc directory, and it's usage will be made clear in the following sections.

There are several options when it comes to transaction management, and they are:

### B.3.1. Auto Commit Mode

When configuring the `DataSource` or the `Connection` to use `autoCommit` (set it to `true`), no transaction management is required. Additional benefit is that any existing Lucene code will work as is with the `JdbcDirectory` (assuming that the `Directory` class was used instead of the actual implementation type).

The main problems with using the `Jdbc` directory in the `autoCommit` mode are: performance suffers because of it, and not all database allow to use `Blobs` with `autoCommit`. As you will see later on, other transaction management are simple to use, and the `Jdbc` directory comes with a set of helper classes that make the transition into a "Jdbc directory enabled code" simple.

### B.3.2. DataSource Transaction Management

When the application does not use any transaction managers (like `JTA` or `Spring's PlatformTransactionManager`), the `Jdbc` directory comes with a simple local transaction management based on `Connection` proxy and thread bound `Connections`.

The `TransactionAwareDataSourceProxy` can wrap a `DataSource`, returning `Jdbc Connection` only if there is no existing `Connection` that was opened before (within the same thread) and not closed yet. Any call to the `close` method on this type of `Connection` (which we call a "not controlled" connection) will result in a `no op`. The `DataSourceUtils#releaseConnection` will also take care and not close the `Connection` if it is not controlled.

So, how do we rollback or commit the `Connection`? `DataSourceUtils` has two methods, `commitConnectionIfPossible` and `rollbackConnectionIfPossible`, which will only commit/rollback the `Connection` if it was proxied by the `TransactionAwareDataSourceProxy`, and it is a controlled `Connection`.

A simple code that performs the above mentioned:

```
JdbcDirectory jdbcDir = // ... create the jdbc directory
Connection conn = DataSourceUtils.getConnection(dataSource);
try {
    IndexReader indexReader = new IndexReader(jdbcDir); // you can also use an already open IndexReader
    // ...
    DataSourceUtils.commitConnectionIfPossible(conn); // will commit the connection if controlling it
} catch (IOException e) {
    DataSourceUtils.safeRollbackConnectionIfPossible(conn);
    throw e;
} finally {
    DataSourceUtils.releaseConnection(conn);
}
```

Note, that the above code will also work when you do have a transaction manager (as described in the next section), and it forms the basis for the `DirectoryTemplate` (described later) that comes with `Jdbc` directory.

### B.3.3. Using External Transaction Manager

For environments that use external transaction managers (like `JTA` or `Spring PlatformTransactionManager`), the transaction management should be performed outside of the code that use the `Jdbc` directory. Do not use `Jdbc` directory `TransactionAwareDataSourceProxy`.

For `JTA` for example, if `Container Managed` transaction is used, the executing code should reside within it. If not, `JTA` transaction should be executed programmatically.

When using `Spring`, the executing code should reside within a transactional context, using either transaction proxy (`AOP`), or the `PlatformTransactionManager` and the `TransactionTemplate` programmatically. **IMPORTANT:** When using `Spring`, you should wrap the `DataSource` with `Spring's` own `TransactionAwareDataSourceProxy`.

### B.3.4. DirectoryTemplate

Since transaction management might require specific code to be written, Jdbc directory comes with a `DirectoryTemplate` class, which allows writing `Directory` implementation and transaction management vanilla code. The directory template perform transaction management support code only if the `Directory` is of type `JdbcDirectory` and the transaction management is a local one (Data Source transaction management).

Each directory based operation (done by Lucene `IndexReader`, `IndexSearcher` and `IndexWriter`) should be wrapped by the `DirectoryTemplate`. An example of using it:

```
DirectoryTemplate template = new DirectoryTemplate(dir); // use a pre-configured directory
template.execute(new DirectoryTemplate.DirectoryCallbackWithoutResult() {
    protected void doInDirectoryWithoutResult(Directory dir) throws IOException {
        IndexWriter writer = new IndexWriter(dir, new SimpleAnalyzer(), true);
        // index write operations
        write.close();
    }
});

// or, for example, if we have a cached IndexSearcher

template.execute(new DirectoryTemplate.DirectoryCallbackWithoutResult() {
    protected void doInDirectoryWithoutResult(Directory dir) throws IOException {
        // indexSearcher operations
    }
});
```

## B.4. File Entry Handler

A `FileEntryHandler` is an interface used by the Jdbc directory to delegate file level operations to it. The `JdbcDirectorySettings` has a default file entry handler which handles all unmapped file names. It also provides the ability to register a `FileEntryHandler` against either an exact file name, or a file extension (3 characters after the '.').

When the `JdbcDirectory` is created, all the different file entry handlers that are registered with the directory settings are created and configured. They will than be used to handle files based on the file names.

When registering a new file entry handler, it must be registered with `JdbcFileEntrySettings`. The `JdbcFileEntrySettings` is a fancy wrapper around java `Properties` in order to provide an open way for configuring file entry handlers. When creating a new `JdbcFileEntrySettings` it already has sensible defaults (refer to the javadoc for them), but of course they can be changed. One important configuration setting is the type of the `FileEntryHandler`, which should be set under the constant setting name: `JdbcFileEntrySettings#FILE_ENTRY_HANDLER_TYPE` and should be the fully qualified class name of the file entry handler.

The Jdbc directory package comes with three different `FileEntryHandler`s. They are:

**Table B.5. File Entry Handler Types**

Type	Description
<code>org.apache.lucene.store.jdbc.handler.NoOpFileEntryHandler</code>	Performs no operations.
<code>org.apache.lucene.store.jdbc.handler.ActualDeleteFileEntryHandler</code>	Performs actual delete from the database when the different delete operations are called. Also support configurable <code>IndexInput</code> and

Type	Description
	IndexOutput (described later).
org.apache.lucene.store.jdbc.handler. MarkDeleteFileEntryHandler	Marks entries in the database as deleted (using the deleted column) when the different delete operations are called. Also support configurable IndexInput and IndexOutput (described later).

Most of the files use the `MarkDeleteFileEntryHandler`, since there might be other currently open `IndexReaders` or `IndexSearchers` that use the files. The `JdbcDirectory` provide the `deleteMarkDeleted()` and `deleteMarkDeleted(delta)` to actually purge old entries that are marked as deleted. It should be scheduled and executed once in a while in order to keep the database table compact.

When creating new `JdbcDirectorySettings`, it already registers different file entry handlers for specific files automatically. For example, the `deleted` file is registered against a `NoOpFileEntryHandler` since we will always be able to delete entries from the database (the `deleted` file is used to store files that could not be deleted from the file system). This results in better performance since no operations are executed against the `deleted` (or `deleted` related files). Another example, is registering the `ActualDeleteFileEntryHandler` against the `segments` file, since we do want to delete it and replace it with a new one when it is written.

### B.4.1. IndexInput Types

Each file entry handler can be associated with an implementation of `IndexInput`. Setting the `IndexInput` should be set under the constant `JdbcFileEntrySettings#INDEX_INPUT_TYPE_SETTING` and be the fully qualified class name of the `IndexInput` implementation.

The Jdbc directory comes with the following `IndexInput` types:

**Table B.6. Index Input Types**

Type	Description
org.apache.lucene.store.jdbc.index. FetchOnOpenJdbcIndexInput	Fetches and caches all the binary data from the database when the <code>IndexInput</code> is opened. Perfect for small sized file entries (like the <code>segments</code> file).
org.apache.lucene.store.jdbc.index. FetchOnBufferReadJdbcIndexInput	Extends the <code>JdbcBufferedIndexInput</code> class, and fetches the data from the database every time the internal buffer need to be refilled. The <code>JdbcBufferedIndexInput</code> allows setting the buffer size using the <code>JdbcBufferedIndexInput#BUFFER_SIZE_SETTING</code> . Remember, that you can set different buffer size for different files by registering different file entry handlers with the <code>JdbcDirectorySettings</code> .
org.apache.lucene.store.jdbc.index. FetchPerTransactionJdbcIndexInput	Caches blobs per transaction. Only supported for dialects that supports blobs per transaction. Note, using this index input requires calling the <code>FetchPerTransactionJdbcIndexInput#releaseBlobs(java.sql.Connection)</code> when the transaction ends. It is automatically taken care of if using <code>TransactionAwareDataSourceProxy</code> . If using JTA for example, a transaction synchronization should be registered with JTA to clear the blobs. Extends the <code>JdbcBufferedIndexInput</code> class, and fetches the data from the database every time the internal buffer need to be

Type	Description
	refilled. The <code>JdbcBufferedIndexInput</code> allows setting the buffer size using the <code>JdbcBufferedIndexInput#BUFFER_SIZE_SETTING</code> . Remember, that you can set different buffer size for different files by registering different file entry handlers with the <code>JdbcDirectorySettings</code> .

The `JdbcDirectorySettings` automatically registers sensible defaults for the default file entry handler and specific ones for specific files. Please refer to the javadocs for the defaults.

## B.4.2. IndexOutput Types

Each file entry handler can be associated with an implementation of `IndexOutput`. Setting the `IndexOutput` should be set under the constant `JdbcFileEntrySettings#INDEX_OUTPUT_TYPE_SETTING` and be the fully qualified class name of the `IndexOutput` implementation.

The Jdbc directory comes with the following `IndexOutput` types:

**Table B.7. Index Output Types**

Type	Description
<code>org.apache.lucene.store.jdbc.index.RAMJdbcIndexOutput</code>	Extends the <code>JdbcBufferedIndexOutput</code> class, and stores the data to be written in memory (within a growing list of <code>bufferSize</code> sized byte arrays). The <code>JdbcBufferedIndexOutput</code> allows setting the buffer size using the <code>JdbcBufferedIndexOutput#BUFFER_SIZE_SETTING</code> . Perfect for small sized file entries (like the segments file).
<code>org.apache.lucene.store.jdbc.index.FileJdbcIndexOutput</code>	Extends the <code>JdbcBufferedIndexOutput</code> class, and stores the data to be written in a temporary file. The <code>JdbcBufferedIndexOutput</code> allows setting the buffer size using the <code>JdbcBufferedIndexOutput#BUFFER_SIZE_SETTING</code> (a write is performed every time the buffer is flushed).
<code>org.apache.lucene.store.jdbc.index.RAMAndFileJdbcIndexOutput</code>	A special index output, that first starts with a RAM based index output, and if a configurable threshold is met, switches to file based index output. The threshold setting can be configured using <code>RAMAndFileJdbcIndexOutput#INDEX_OUTPUT_THRESHOLD_SETTING</code> .

The `JdbcDirectorySettings` automatically registers sensible defaults for the default file entry handler and specific ones for specific files. Please refer to the javadocs for the defaults.