Compass

Reference Documentation

2.0.2

Table of Contents

Preface .....
1. Introduction .....
1.1. Overview 1
1.2. I use 2
1.2.1. ... Lucene 2
1.2.2. ... Domain Model 4
1.2.3. ... Xml Model 4
1.2.4. ... No Model 4
1.2.5. ... ORM Framework 4
1.2.6. ... Spring Framework 5
I. Compass Core .....
2. Introduction .....
2.1. Overview 7
2.2. Session Lifecycle 8
2.3. Template and Callback 8
3. Configuration .....
3.1. Programmatic Configuration 9
3.2. XML Configuration 11
3.2.1. Schema Based Configuration 11
3.2.2. DTD Based Configuration 13
3.3. Obtaining a Compass reference 14
3.4. Configuring Callback Events 14
4. Connection .....
4.1. File System Store 15
4.2. RAM Store 16
4.3. Jdbc Store 16
4.3.1. Managed Environment 16
4.3.2. Data Source Provider 17
4.3.3. File Entry Handler 18
4.3.4. DDL 19
4.4. Lock Factory 19
4.5. Local Directory Cache 20
4.6. Lucene Directory Wrapper 20
4.6.1. SyncMemoryMirrorDirectoryWrapperProvider 21
4.6.2. AsyncMemoryMirrorDirectoryWrapperProvider 21
5. Search Engine .....
5.1. Introduction 23
5.2. Alias, Resource and Property 23
5.2.1. Using Resource/Property 23
5.3. Analyzers 23
5.3.1. Configuring Analyzers 24
5.3.2. Analyzer Filter 25
5.3.3. Handling Synonyms 25
5.4. Query Parser 26
5.5. Index Structure 26
5.6. Transaction 27
5.6.1. Locking 27
5.6.2. Isolation 28

5.6.3. Transaction Log 29
5.7. All Support 30
5.8. Sub Index Hashing 31
5.8.1. Constant Sub Index Hashing 31
5.8.2. Modulo Sub Index Hashing 32
5.8.3. Custom Sub Index Hashing 34
5.9. Optimizers 35
5.9.1. Scheduled Optimizers 35
5.9.2. Aggressive Optimizer 35
5.9.3. Adaptive Optimizer 35
5.9.4. Null Optimizer 36
5.10. Merge 36
5.10.1. Merge Policy 36
5.10.2. Merge Scheduler 36
5.11. Index Deletion Policy 36
5.12. Spell Check / Did You Mean 37
5.12.1. Spell Index 38
5.13. Direct Lucene 39
5.13.1. Wrappers 39
5.13.2. Searcher And IndexReader 39
6. OSEM - Object/Search Engine Mapping .....
6.1. Introduction 41
6.2. Searchable Classes 41
6.2.1. Alias 42
6.2.2. Root 43
6.2.3. Sub Index 43
6.3. Searchable Class Mappings 43
6.3.1. Searchable Id and Searchable Meta Data 43
6.3.2. Searchable Id Component 44
6.3.3. Searchable Parent 45
6.3.4. Searchable Property and Searchable Meta Data 45
6.3.5. Searchable Constant 46
6.3.6. Searchable Dynamic Meta Data 47
6.3.7. Searchable Reference 47
6.3.8. Searchable Component 48
6.3.9. Searchable Cascade 49
6.3.10. Searchable Analyzer 49
6.3.11. Searchable Boost 50
6.4. Specifics 50
6.4.1. Handling Collection Types 50
6.4.2. Managed Id 50
6.4.3. Handling Inheritance 51
6.4.4. Polymorphic Relationships 52
6.4.5. Cyclic Relationships 53
6.4.6. Annotations and Xml Combined 53
6.4.7. Support Unmarshall 54
6.4.8. Configuration Annotations 54
6.5. Searchable Annotations Reference 54
6.6. Searchable Xml Reference 54
6.6.1. compass-core-mapping 54
6.6.2. class 55
6.6.3. contract 56

6.6.4. id 57
6.6.5. property 58
6.6.6. analyzer 59
6.6.7. boost 60
6.6.8. meta-data 60
6.6.9. dynamic-meta-data 61
6.6.10. component 63
6.6.11. reference 64
6.6.12. parent 65
6.6.13. constant 65
7. XSEM - Xml to Search Engine Mapping .....
7.1. Introduction 67
7.2. Xml Object 67
7.3. Xml Content Handling 67
7.4. Raw Xml Object 69
7.5. Mapping Definition 69
7.5.1. xml-object 70
7.5.2. xml-id 71
7.5.3. xml-property 72
7.5.4. xml-analyzer 73
7.5.5. xml-boost 74
7.5.6. xml-content 74
8. RSEM - Resource/Search Engine Mapping .....
8.1. Introduction 76
8.2. Mapping Declaration 76
8.2.1. resource 77
8.2.2. resource-contract 78
8.2.3. resource-id 78
8.2.4. resource-property 79
8.2.5. resource-analyzer 80
8.2.6. resource-boost 81
9. Common Meta Data .....
9.1. Introduction 82
9.2. Commnon Meta Data Definition 82
9.3. Using the Definition 83
9.4. Commnon Meta Data Ant Task 84
10. Transaction .....
10.1. Introduction 85
10.2. Session Lifecycle 85
10.3. Local Transaction 85
10.4. JTA Synchronization Transaction 85
10.5. XA Transaction 86
11. Working with objects .....
11.1. Introduction 87
11.2. Making Object/Resource Searchable 87
11.3. Loading an Object/Resource 87
11.4. Deleting an Object/Resource 87
11.5. Searching 88
11.5.1. Query String Syntax 88
11.5.2. Query String - Range Queries Extensions 89
11.5.3. CompassHits, CompassDetachedHits & CompassHitsOperations 89
11.5.4. CompassQuery and CompassQueryBuilder 89

11.5.5. Terms and Frequencies 91
11.5.6. CompassSearchHelper 91
11.5.7. CompassHighlighter 91
II. Compass Vocabulary .....
12. Introduction .....
13. Dublin Core .....
III. Compass Gps .....
14. Introduction .....
14.1. Overview 97
14.2. CompassGps 97
14.2.1. SingleCompassGps 97
14.2.2. DualCompassGps 98
14.3. CompassGpsDevice 98
14.3.1. MirrorDataChangesGpsDevice 98
14.4. Programmatic Configuration 99
14.5. Parallel Device 99
14.6. Building a Gps Device 101
15. JDBC .....
15.1. Introduction 103
15.2. Mapping 103
15.2.1. ResultSet Mapping 103
15.2.2. Table Mapping 104
15.3. Mapping - MirrorDataChanges 105
15.3.1. ResultSet Mapping 105
15.3.2. Table Mapping 105
15.3.3. Jdbc Snapshot 105
15.4. Resource Mapping 106
15.5. Putting it All Together 106
16. Embedded Hibernate .....
16.1. Introduction 107
16.2. Configuration 108
16.3. Transaction Management 108
17. Hibernate .....
17.1. Introduction 109
17.2. Configuration 109
17.2.1. Deprecated Hibernate Devices 109
17.3. Index Operation 110
17.4. Real Time Data Mirroring 110
17.5. HibernateSyncTransaction 110
17.6. Hibernate Transaction Interceptor 111
18. JPA (Java Persistence API) .....
18.1. Introduction 112
18.2. Configuration 112
18.3. Index Operation 112
18.4. Real Time Data Mirroring 113
19. Embedded OpenJPA .....
19.1. Introduction 114
19.2. Configuration 114
19.3. Index Operation 114
19.4. Real Time Data Mirroring 115
19.5. OpenJPA Helper 115
20. Embedded TopLink Essentials .....

20.1. Introduction 116
20.2. Configuration 116
20.3. Transaction Management 117
21. Embedded EclipseLink .....
21.1. Introduction 118
21.2. Configuration 118
21.3. Transaction Management 119
22. JDO (Java Data Objects) .....
22.1. Introduction 120
22.2. Configuration 120
22.3. Index Operation 120
22.4. Real Time Data Mirroring 120
23. OJB (Object Relational Broker) .....
23.1. Introduction 121
23.2. Index Operation 121
23.3. Real Time Data Mirroring 121
23.4. Configuration 121
24. iBatis .....
24.1. Introduction 123
24.2. Index Operation 123
24.3. Configuration 123
IV. Compass Spring .....
25. Introduction .....
25.1. Overview 125
25.2. Compass Definition in Application Context 125
26. DAO Support .....
26.1. Dao and Template 127
27. Spring Transaction .....
27.1. Introduction 128
27.2. LocalTransaction 128
27.3. JTASyncTransaction 128
27.4. SpringSyncTransaction 128
27.5. CompassTransactionManager 128
28. Hibernate 3 Gps Device Support .....
28.1. Deprecation Note 129
28.2. Introduction 129
28.3. SpringHibernate3GpsDevice 129
29. OJB Gps Device Support .....
29.1. Introduction 130
29.2. SpringOjbGpsDevice 130
29.3. SpringOjbGpsDeviceInterceptor 130
30. Jdbc Gps Device Support .....
30.1. Introduction 131
30.2. ResultSet Mapping 131
30.3. Table Mapping 133
31. Spring AOP .....
31.1. Introduction 135
31.2. Advices 135
31.3. Dao Sample 135
31.4. Transactional Serivce Sample 136
32. Spring MVC Support .....
32.1. Introduction 138

32.2. Support Classes 138
32.3. Index Controller 138
32.4. Search Controller 138
V. Compass Needle .....
33. GigaSpaces .....
33.1. Overview 140
33.2. Lucene Directory 140
33.3. Compass Store 140
33.4. Searchable Space 141
34. Coherence .....
34.1. Overview 142
34.2. Lucene Directory 142
34.3. Compass Store 142
35. Terracotta .....
35.1. Overview 144
35.2. Lucene Directory 144
35.3. Compass Store 144
VI. Compass Samples .....
36. Library Sample .....
36.1. Introduction 147
36.2. Running The Sample 147
37. Petclinic Sample .....
37.1. Introduction 148
37.2. Running The Sample 148
37.3. Data Model In Petclinic 148
37.3.1. Common Meta-data (Optional) 148
37.3.2. Resource Mapping 149
37.3.3. OSEM 150
37.4. Data Access In Petclinic 150
37.4.1. Hibernate 150
37.4.2. Apache OJB 150
37.4.3. JDBC 150
37.5. Web (MVC) in Petclinic 151
VII. Appendixes .....
A. Configuration Settings .....
A.1. Compass Configuration Settings 153
A.1.1. compass.engine.connection 153
A.1.2. JNDI 153
A.1.3. Property 154
A.1.4. Transaction Level 154
A.1.5. Transaction Strategy 155
A.1.6. Property Accessor 156
A.1.7. Converters 157
A.1.8. Search Engine 160
A.1.9. Search Engine Jdbc 163
A.1.10. Search Engine Analyzers 166
A.1.11. Search Engine Analyzer Filters 167
A.1.12. Search Engine Highlighters 167
A.1.13. Other Settings 169
B. Lucene Jdbc Directory .....
B.1. Overview 170
B.2. Performance Notes 172

B.3. Transaction Management 172
B.3.1. Auto Commit Mode 172
B.3.2. DataSource Transaction Management 173
B.3.3. Using External Transaction Manager 173
B.3.4. DirectoryTemplate 174
B.4. File Entry Handler 174
B.4.1. IndexInput Types 175
B.4.2. IndexOutput Types 176

Preface

Compass Goal

"Simplify the integration of Search Engine into any application"

As you will see, Compass is geared towards integrating Search Engine functionality into any type of application (web app, rich client, middle-ware, ...). We ask you, the user, to give feedback on how complex it was to integrate Compass into your application, and places where Compass can be enhanced to make things even simpler.

Compass is a powerful, transactional Java Search Engine framework. Compass allows you to declaratively map your Object domain model to the underlying Search Engine, synchronizing data changes between search engine 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 different datasources.
- Write less code, find data quicker.

This document provides a reference guide to Compass's features. Since this document is still to be considered very much work-in-progress, if you have any requests or comments, please post them on Compass forum, or Compass JIRA (issue tracking).

Before we continue, the Compass team would like to thank the Hibernate and Spring Framework teams, for providing the template DocBook definition and configuration, which help us create this reference guide.

Chapter 1. Introduction

History

Shay Banon (kimchy), the creator of Compass, decided to write a simple Java based recipe management software for his wife (who happens to be a chef). Main requirement for the tool, since it was going to hold substantial cooking knowledge, was to be able to get to the relevant information fast. Using Spring Framework, Hibernate, and all the other tools out there that makes a developer life simple, he was surprised to find nothing similar in the search engine department. Now, don't get him wrong, Lucene is an amazing search engine (or IR library), but developers want simplicity, and Lucene comes with an added complexity that caused Shay to start Compass.

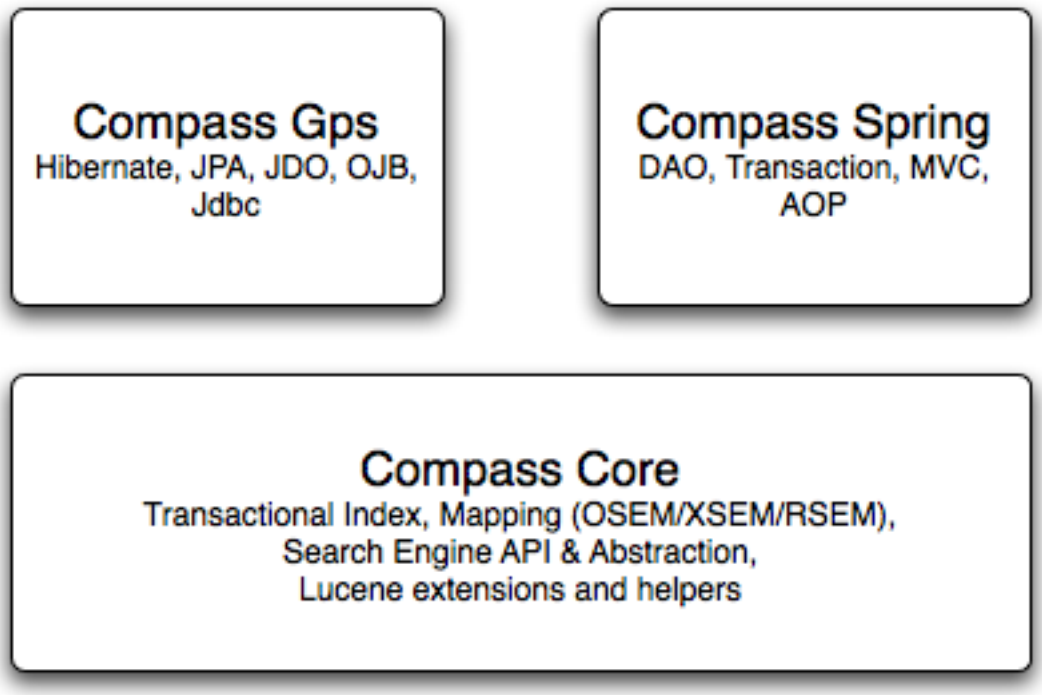
In todays applications, search is becoming a "must have" requirement. Users expect applications (rich clients, web based, sever side, ...) to provide snappy and relevant search results the same way Google does for the web. Let it be a recipe management software, a trading application, or a content management driven web site, users expect search results across the whole app business domain model.

Java developers on the other hand, need to implement it. As Java developers are getting used to simplified development model, with Hibernate, Spring Framework, and EJB3 to name a few, up until now there was a lack in a simple to use Java Search Engine solution. Compass aim is to fill this gap.

Many applications, once starting to use a search engine in order to implement that illusive search box, find that the search engine can then be used for many data extraction related operations. Once a search engine holds a valid representation of the application business model, many times it just makes sense to execute simple queries against it instead of going to the actual data store (usually a database). Two prime examples are Jira and Confluence, which perform many of the reporting and search (naturally) operations using a search engine instead of the usual database operations.

1.1. Overview

Compass provides a breadth of features geared towards integrating search engine functionality. The next diagram shows the different Compass modules, followed by a short description of each one.



Overview of Compass

Compass Core is the most fundamental part of Compass. It holds Lucene extensions for transactional index, search engine abstraction, ORM like API, transaction management integration, different mappings technologies (OSEM, XSEM and RSEM), and more. The aim of Compass core is to be usable within different scenarios and environments, and simplify the core operations done with a search engine.

Compass Gps aim is to integrate with different content sources. The prime feature is the integration with different ORM frameworks (Hibernate, JPA, JDO, OJB), allowing for almost transparent integration between a search engine and an ORM view of content that resides in a database. Other features include a Jdbc integration, which allows to index database content using configurable SQL expression responsible for extracting the content.

Compass Spring integrate Compass with the Spring Framework. Spring, being an easy to use application framework, provides a simpler development model (based on dependency injection and much more). Compass integrates with Spring in the same manner ORM Frameworks integration is done within the Spring Framework code-base. It also integrates with Spring transaction abstraction layer, AOP support, and MVC library.

1.2. I use ...

The following sections are aimed to be a brief introduction and a navigation map for people who are familiar or use this different technologies:

1.2.1. ... Lucene

Direct Lucene

Compass tries to be a good Lucene citizen, allowing to use most of Lucene classes directly within Compass. If your application has a specialized Query, Analyzer, or Filter, you can use them directly with Compass. Compass does have its own index structure, divided into sub indexes, but each sub index is a fully functional

Lucene index.

Search Engine Abstraction

Compass created a search engine abstraction, with its main (and only) implementation using Lucene. Lucene is an amazing, fast, and stable search engine (or IR library), yet the main problem with integrating Lucene with our application is its low-level usage and API.

For people who use or know Lucene, it is important to explain new terms that are introduced by Compass. `Resource` is Compass abstraction on top of a Lucene `Document`, and `Property` is Compass abstraction on top of Lucene `Field`. Both do not add much on top of the actual Lucene implementations, except for `Resource`, which is associated with an *Alias*. For more information, please read Chapter 5, *Search Engine*.

RSEM - Resource/Search Engine Mapping

`Resource` is the lowest level data object used in Compass, with all different mapping technologies are geared towards generating it. Compass comes with a low level mapping technology called *RSEM* (Resource/Search Engine Mapping), which allows to declaratively define resource mapping definitions. RSEM can be used when an existing system already uses Lucene (upgrade to Compass should be minimal), or when an application does not have a rich domain model (Object or XML).

An additional feature built on top of Compass converter framework, is that a `Property` value does not have to be a String (as in Lucene `Field`). Objects can be used as values, with specific or default converters applied to them. For more information, please read Chapter 8, *RSEM - Resource/Search Engine Mapping*.

Simple API

Compass exposes a very simple API. If you have experience with an ORM tool (Hibernate, JPA, ...), you should feel very comfortable with Compass API. Also, Lucene has three main classes, `IndexReader`, `Searcher` and `IndexWriter`. It is difficult, especially for developers unfamiliar with Lucene, to understand how to perform operations against the index (while still having a performant system). Compass has a single interface, with all operations available through it. Compass also abstract the user from the gory details of opening and closing readers/searchers/writers, as well as caching and invalidating them. For more information, please read Chapter 2, *Introduction*, and Chapter 11, *Working with objects*.

Transactional Index and Integration

Lucene is not transactional. This causes problems when trying to integrate Lucene with other transactional resources (like database or messaging). Compass provides support for two phase commits transactions (read_committed and serializable), implemented on top of Lucene index segmentations. The implementation provides fast commits (faster than Lucene), though they do require the concept of Optimizers that will keep the index at bay. For more information, please read Section 5.6, “Transaction”, and Section 5.9, “Optimizers”.

On top of providing support for a transactional index, Compass provides integration with different transaction managers (like JTA), and provides a local one. For more information, please read Chapter 10, *Transaction*.

Fast Updates

In Lucene, in order to perform an update, you must first delete the old `Document` and then create a new `Document`. This is not trivial, especially because of the usage of two different interfaces to perform the delete (IndexReader) and create (IndexWriter) operations, it is also very delicate in terms of performance. Thanks to Compass support for transactional index, and the fact that each saved `Resource` in Compass must be identifiable (through the use of mapping definition), makes executing an update using Compass both simple

(the operation is called `save`), and fast.

All Support

When working with Lucene, there is no way to search on all the fields stored in a Document. One must programmatically create synthetic fields that aggregate all the other fields in order to provide an "all" field, as well as providing it when querying the index. Compass does it all for you, by default Compass creates that "all" field and it acts as the default search field. Of course, in the spirit of being as configurable as possible, the "all" property can be enabled or disabled, have a different name, or not act as the default search property. One can also exclude certain mappings from participating in the all property.

Index Fragmentation

When building a Lucene enabled application, sometimes (for performance reasons) the index actually consists of several indexes. Compass will automatically fragment the index into several sub indexes using a configurable sub index hashing function, allowing to hash different searchable objects (`Resource`, mapped object, or an `XmlObject`) into a sub index (or several of them). For more information, please read Section 5.5, “Index Structure”.

1.2.2. ... Domain Model

One of Compass main features is OSEM (Object/Search Engine Mapping). Using either annotations or xml definitions (or a combination), mapping definitions from a rich domain model into a search engine can be defined. For more information, please read Chapter 6, *OSEM - Object/Search Engine Mapping*.

1.2.3. ... Xml Model

One of Compass main features is XSEM (Xml/Search Engine Mapping). If your application is built around Xml data, you can map it directly to the search engine using simple xml based mapping definitions based on xpath expressions. For more information, please read Chapter 7, *XSEM - Xml to Search Engine Mapping*.

1.2.4. ... No Model

If no specific domain model is defined for the application (for example, in a messaging system based on properties), RSEM (Resource/Search Engine Mapping) can be used. A `Resource` can be considered as a fancy hash map, allowing for completely open data that can be saved in Compass. A resource mapping definition needs to be defined for "types" of resources, with at least one resource id definition (a resource must be identifiable). Additional resource properties mapping can be defined, with declarative definition of its characteristics (search engine, converter, ...). For more information, please read Chapter 8, *RSEM - Resource/Search Engine Mapping*.

1.2.5. ... ORM Framework

Built on top of Compass Core, Compass Gps (which is aimed at integrating Compass with other datasources) integrates with most popular ORM frameworks. The integration consists of two main features:

Index Operation

Automatically index data from the database using the ORM framework into the search engine using Compass (and OSEM). Objects that have both OSEM and ORM definitions will be indexed, with the ability to provide custom filters.

Mirror Operation

For ORM frameworks that support event registration (most do), Compass will automatically register its own event listeners to reflect any changes made to the database using the ORM API into the search engine.

For more information, please read Chapter 14, *Introduction*. Some of the ORM frameworks supports are: Chapter 16, *Embedded Hibernate*, Chapter 18, *JPA (Java Persistence API)*, Chapter 19, *Embedded OpenJPA*, Chapter 22, *JDO (Java Data Objects)*, Chapter 23, *OJB (Object Relational Broker)* and Chapter 24, *iBatis*.

1.2.6. ... Spring Framework

The aim of Compass::Spring module is to provide seamless integration with the Spring Framework (as if a Spring developer wrote it :)).

First level of integration is very similar to Spring provided ORM integration, with a `LocalCompassBean` which allows to configure Compass within a Spring context, and a `CompassDaoSupport` class. For more information, please read Chapter 25, *Introduction* and Chapter 26, *DAO Support*.

Spring AOP integration, providing simple 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. For more information, please read Chapter 31, *Spring AOP*.

Spring `PlatformTransactionManager` abstraction integration, using its `SpringSyncTransactionFactory` to register synchronization with Spring on going transaction. This allows Compass to work in environments where Spring specific transactions managers are used, like `HibernateTransactionManager`. For more information, please read Chapter 27, *Spring Transaction*.

For web applications that use Spring MVC, Compass provides a search and index controllers. The index controller can automatically perform the index operation on a `CompassGps`, only the initiator view and result view need to be written. The search controller can automatically perform the search operation (With pagination), requiring only the search initiator and search results view (usually the same one). For more information, please read Chapter 32, *Spring MVC Support*.

Last, `LocalCompassBean` can be configured using Spring 2 new schema based configuration.

Part I. Compass Core

Compass Core provides the core of Compass simplification of search engine integration into an application. Compass is very simple to use, and you can start/enable a searchable application in a matter of hours, many times without knowing about Compass advance features and extendibility.

The two most important chapters are Chapter 2, *Introduction*, which explains the high level Compass API's (similar to your usual ORM framework), and Chapter 3, *Configuration* which explains how to configure a Compass instance. Chapter 5, *Search Engine* dives into details of Compass search engine abstraction, explaining concepts, index structure, and Lucene extensions (both Compass extensions and Compass enabling Lucene features simply). In order to start using Compass, reading this chapter is not required, though it does provide a good overview of how Compass abstracts Lucene, and explains how to configure advance search engine related features (Analyzers, Optimizers, Sub Index Hashing, and so on).

The following chapters dive into details of Compass different mapping technologies. Chapter 6, *OSEM - Object/Search Engine Mapping* explain how to use Compass OSEM (Object/Search Engine Mapping), Chapter 7, *XSEM - Xml to Search Engine Mapping* goes into details of how to use Compass XSEM (Xml/Search Engine Mapping), and Chapter 8, *RSEM - Resource/Search Engine Mapping* dives into Compass RSEM (Resource/Search Engine Mapping), which is a low level, Lucene like, mapping support.

Chapter 9, *Common Meta Data* explains Compass support for creating a semantic model defined outside of the mapping definitions. Using it is, in the spirit of Compass, completely optional, and depends on the developers if they wish to use it within their Compass enabled application.

Chapter 10, *Transaction* goes into details of the different ways to integrate Compass transaction support within different transaction managers. It explains both local (Compass managed) transaction, and JTA integration.

Chapter 2. Introduction

2.1. Overview

Compass API

As you will learn in this chapter, Compass high level API looks strangely familiar. If you used an ORM framework (Hibernate, JDO or JPA), you should feel right at home. This is of-course, intentional. The aim is to let the developer learn as less as possible in terms of interaction with Compass. Also, there are so many design patterns of integrating this type of API with different applications models, that it is a shame that they won't be used with Compass as well.

For Hibernate users, `Compass` maps to `SessionFactory`, `CompassSession` maps to `Session`, and `CompassTransaction` maps to `Transaction`.

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

Compass provide 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. 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 (it is non thread safe). With a `CompassSession`, applications can save and retrieve any searchable 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, they must be specified using either OSEM or XSEM (with XSEM `XmlObject`). In order to work with Resources, they must be specified using RSEM (Resource can still be used with OSEM/RSEM to display search results, since Objects/Xml end up being converted to Resources). Compass will then retrieve the declared searchable data from the Object automatically when saving Objects within Compass. When querying the Search Engine, Compass 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, lets 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.resource(0);
    ...
    tx.commit();
} catch (CompassException ce) {
    if (tx != null) tx.rollback();
} finally {
    session.close();
}
```

2.2. 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, Spring 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.

When using the `openSession` method, Compass will automatically try and join an already running outer transaction. An outer transaction can be an already running local Compass transaction, a JTA transaction, or a Spring managed transaction. If Compass manages to join an existing outer transaction, the application does not need to call `CompassSession#beginTransaction()` or use `CompassTransaction` to manage the transaction (since it is already managed). This allows to simplify the usage of Compass within managed environments (CMT or Spring) where a transaction is already in progress by not requiring explicit Compass code to manage a Compass transaction.

2.3. Template and Callback

Compass 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 3. Configuration

Configuration Samples

Throughout this manual, we will use the schema based configuration file to show examples of how to configure certain features. This does not mean that they can not be expressed in a settings based configuration (either programmatic or DTD based configuration file). For a complete list of all the different settings in compass, please refer to Appendix A, *Configuration Settings*.

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 internally and with the underlying Search Engine. This section describes the configuration API.

In order to create a `Compass` instance, it first must be configured. `CompassConfiguration` can be used in order to configure a `Compass` instance, by having the ability to add different mapping definitions, configure Compass based on xml configuration files, and expose a programmatic configuration options.

For Java 5 based applications (mainly ones that use OSEM annotations), `CompassAnnotationsConfiguration` can be used (which extends `CompassConfiguration`). For simplicity, Compass comes with `CompassConfigurationFactory`, which tries to be smart and detect based on the JVM version and the included compass modules, which configuration to create. Here is an example of the preferred way to obtain a `CompassConfiguration` instance:

```
CompassConfiguration conf =
      CompassConfigurationFactory.newConfiguration();
```

3.1. Programmatic Configuration

A `Compass` instance can be programatically configured using `CompassConfiguration`. Two main configuration aspects are adding mapping definitions, and setting different settings.

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

Table 3.1. Configuration Mapping API

API	Description
`addFile(String)`	Loads the mapping file (`cpm` or `cmd`) according to the specified file path string.
`addFile(File)`	Loads the mapping file (`cpm` or `cmd`) according to the specified file object reference.
`addClass(Class)`	Loads the mapping file (`cpm`) according to the specified class. `test.Author.class` will map to `test/Author.cpm.xml` within the class path. Can also add annotated classes if using Compass annotations support.

API	Description
`addURL(URL)`	Loads the mapping file (`cpm` or `cmd`) according to the specified URL.
`addResource(String)`	Loads the mapping file (`cpm` or `cmd`) according to the specified resource path from the class path.
`addInputStream(InputStream)`	Loads the mapping file (`cpm` or `cmd`) according to the specified input stream.
`addDirectory(String)`	Loads all the files named `*.cpm.xml` or `*.cmd.xml` from within the specified directory.
`addJar(File)`	Loads all the files named `*.cpm.xml` or `*.cmd.xml` from within the specified Jar file.
`addScan(String basePackage, String pattern)`	Scans for all the mappings that exist wihtin the base backage recursively. An optioal ant style pattern can be provided as well. The mappings detected are all the xml based mappings. Annotation based mappings will be detected automatically if either ASM or Javassist exists within the classpath.
`addMappingResolver(MappingResolver)`	Uses a class that implements the `MappingResolver` to get an `InputStream` for xml mapping definitions.

Other than mapping file configuration API, Compass 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 Appendix A, *Configuration Settings*.

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 names).

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);
```

An important aspect of settings (properties like) configuration is the notion of group settings. Similar to the way log4j properties configuration file works, different aspects of Compass need to be configured in a grouped nature. If we take Compass converter configuration as an example, here is an example of a set of settings to configure a custom converter called `test`:

```
org.compass.converter.test.type=eg.TestConverter
org.compass.converter.test.param1=value1
org.compass.converter.test.param2=value2
```

Compass defined prefix for all converter configuration is `org.compass.converter`. The segment that comes afterwards (up until the next '.') is the converter (group) name, which is set to `test`. This will be the converter name that the converter will be registered under (and referenced by in different mapping definitions). Within the group, the following settings are defined: `type`, `param1`, and `param2`. `type` is one of the required settings for a custom Compass converter, and has the value of the fully qualified class name of the converter implementation. `param1` and `param2` are custom settings, that can be used by the custom converter (it should implement `CompassConfigurable`).

3.2. 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 3.2. Compass Configuration API

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

3.2.1. Schema Based Configuration

Schema and Settings

Compass uses the schema based configuration as a different view on top of its support for settings based configuration (properties like). Compass translates all the different, more expressive, xml structure into their equivalent settings as described in Appendix A, *Configuration Settings*.

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 and 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, just to get a feeling of the structure and nature of the configuration file. The first is a simple file based index with the OSEM definitions for the Author class.

```
<compass-core-config xmlns="http://www.compass-project.org/schema/core-config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.compass-project.org/schema/core-config
           http://www.compass-project.org/schema/compass-core-config-2.0.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.compass-project.org/schema/core-config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.compass-project.org/schema/core-config
          http://www.compass-project.org/schema/compass-core-config-2.0.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.compass-project.org/schema/core-config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.compass-project.org/schema/core-config
          http://www.compass-project.org/schema/compass-core-config-2.0.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.compass-project.org/schema/core-config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.compass-project.org/schema/core-config
          http://www.compass-project.org/schema/compass-core-config-2.0.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.compass-project.org/schema/core-config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.compass-project.org/schema/core-config
         http://www.compass-project.org/schema/compass-core-config-2.0.xsd">

  <compass name="default">

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

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

3.2.2. 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 Appendix A, *Configuration Settings*.

And here is an example of the xml configuration file:

```
<!DOCTYPE compass-core-configuration PUBLIC
"-//Compass/Compass Core Configuration DTD 2.0//EN"
"http://www.compass-project.org/dtd/compass-core-configuration-2.0.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>
```

3.3. 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.

3.4. Configuring Callback Events

Compass allows to configure events that will be fired once certain operations occur in using Compass, for example, `save` operation.

Configuring event listener can be done settings. For example, to configure a pre save event listener, the following setting should be used: `compass.event.preSave.mylistener.type` and its value can be the actual class name of the listener.

More information for each listener can be found in the javadoc under the `org.compass.events` package. An important note with regards to pre listener is the fact that they can filter out certain operations.

Chapter 4. Connection

Lucene Directory

Lucene comes with a `Directory` abstraction on top of the actual index storage. Compass uses Lucene built in different directories implementations, as well as have custom implementations built on top of it.

The only required configuration for a `Compass` instance (using the `CompassConfiguration`) is its connection. The connection controls where the index will be saved, or in other words, the storage location of the index. This chapter will review the different options of index storage that comes with Compass, and try to expand on some of important aspects when using a certain storage (like clustering support).

4.1. File System Store

By far the most popular and simple of all storage options is storing the index on the file system. Here is an example of a simple file system based connection configuration that stores the index in the `target/test-index` path:

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

Another option for file system based configuration is using Java 1.4 NIO feature. The NIO should perform better under certain environment/load then the default file based one. We recommend performing some performance tests (preferable as close to your production system configuration as possible), and check which one performs better. Here is an example of a simple file system based connection configuration that stores the index in the `target/test-index` path:

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

When using file system based index storage, locking (for transaction support) is done using lock files. The existence of the file means a certain sub index is locked. The default lock file directory is `java.io.tmp` system property.

Clustering support for file system based storage usually means sharing the file system between different machines (running different Compass instances). Current locking mechanism will require to set the locking directory on the shared file system, here is an example of how to set it:

```
<compass name="default">
  <connection>
      <mmap path="/shared/index-data"/>
  </connection>

  <transaction lockDir="/shared/index-lock" />
</compass>
```

Another important note regarding using a shared file system based index storage is not to use NFS. For best

performance, a SAN based solution is recommended.

4.2. RAM Store

Using the RAM based index store, the index data can be stored in memory. This is usable for fast indexing and searching, on the expense of no long lived storage. Here is an example of how it can be configured:

```
<compass name="default">
  <connection>
      <ram path="/index"/>
  </connection>
</compass>
```

4.3. Jdbc Store

The Jdbc store connection type allows the index data to be stored within a database. The schema used for storing the index actually simulates a file system based tree, with each row in a sub index table representing a "file" with its binary data.

Compass implementation, `JdbcDirectory`, which is built on top of Lucene `Directory` abstraction is completely decoupled from the rest of Compass, and can be used with pure Lucene applications. For more information, please read Appendix B, *Lucene Jdbc Directory*. Naturally, when using it within Compass it allows for simpler configuration, especially in terms of transaction management and Jdbc `DataSource` management.

Here is a simple example of using Jdbc to store the index. The example configuration assumes a standalone configuration, with no data source pooling.

```
<compass name="default">
  <connection>
      <jdbc>
          <dataSourceProvider>
              <driverManager  url="jdbc:hsqldb:mem:test"
                              username="sa" password=""
                              driverClass="org.hsqldb.jdbcDriver" />
          </dataSourceProvider>
      </jdbc>
  </connection>
</compass>
```

The above configuration does not define a dialect attribute on the jdbc element. Compass will try to auto-detect the database dialect based on the database meta-data. If it fails to find one, a dialect can be set, in our case it should be `dialect="org.apache.lucene.store.jdbc.dialect.HSQLDialect"`.

4.3.1. Managed Environment

It is important to understand if Compass is working within a "managed" environment or not when it comes to a Jdbc index storage. A managed environment is an environment where Compass is not in control of the transaction management (in case of configuring Compass with JTA or Spring transaction management). If Compass is in control of the transaction, i.e. using Local transaction factory, it is not considered a managed environment.

When working in a non managed environment, Compass will wrap the data source with a `TransactionAwareDataSourceProxy`, and will commit/rollback the Jdbc connection. When working within a managed environment, no wrapping will be performed, and Compass will let the external transaction manager

to commit/rollback the connection.

Usually, but not always, when working in a managed environment, the Jdbc data source used will be from an external system/configuration. Most of the times it will either be JNDI or external data source provider (like Spring). For more information about different data source providers, read the next section.

By default, Compass works as if within a non managed environment. The `managed` attribute on the `jdbc` element should be set to `true` otherwise.

4.3.2. Data Source Provider

Compass allows for different Jdbc `DataSource` providers. A `DataSourceProvider` implementation is responsible for configuring and providing a Jdbc `DataSource` instance. A data source implementation is very important when it comes to performance, especially in terms of pooling features.

All different data source supported by Compass allow to configure the autoCommit flag. There are three values allowed for autoCommit: `false`, `true` and `external` (don't set the autoCommit explicitly, assume it is configured elsewhere). The autoCommit mode defaults to `false` and it is the recommended value (external can also be used, but make sure to set the actual data source to false).

4.3.2.1. Driver Manager

The simplest of all providers. Does not requires any external libraries or systems. Main drawback is performance, since it performs no pooling of any kind. The first sample of a Jdbc configuration earlier in this chapter used the driver manager as a data source provider.

4.3.2.2. Jakarta Commons DBCP

Compass can be configured to use Jakarta Commons DBCP as a data source provider. It is the preferred option than the driver manager provider for performance reasons (it is up to you if you want to use it or c3p0 explained later in this section). Here is an example of using it:

```
<compass name="default">
  <connection>
      <jdbc>
        <dataSourceProvider>
            <dbcp url="jdbc:hsqldb:mem:test"
                  username="sa" password=""
                  driverClass="org.hsqldb.jdbcDriver"
                  maxActive="10" maxWait="5" maxIdle="2" initialSize="3" minIdle="4"
                  poolPreparedStatements="true" />
        </dataSourceProvider>
      </jdbc>
  </connection>
</compass>
```

The configuration shows the different settings that can be used on the dbcp data source provider. They are, by no means, the recommended values for a typical system. For more information, please consult Jakarta Commons DBCP documentation.

4.3.2.3. c3p0

Compass can be configured using c3p0 as a data source provider. It is the preferred option than the driver manager provider for performance reasons (it is up to you if you want to use it or Jakarta Commons DBCP explained previously in this section). Here is an example of using it:

```
<compass name="default">
  <connection>
```

```
      <jdbc>
        <dataSourceProvider>
          <c3p0 url="jdbc:hsqldb:mem:test"
                username="testusername" password="testpassword"
                driverClass="org.hsqldb.jdbcDriver" />
        </dataSourceProvider>
      </jdbc>
  </connection>
</compass>
```

The c3p0 data source provider will use c3p0 `ComboPooledDataSource`, with additional settings can be set by using c3p0.properties stored as a top-level resource in the same CLASSPATH / classloader that loads c3p0's jar file. Please consult the c3p0 documentation for additional settings.

4.3.2.4. JNDI

Compass can be configured to look up the data source using JNDI. Here is an example of using it:

```
<compass name="default">
  <connection>
      <jdbc>
        <dataSourceProvider>
          <jndi lookup="testds" username="testusername" password="testpassword" />
        </dataSourceProvider>
      </jdbc>
  </connection>
</compass>
```

The jndi lookup environment, including the `java.naming.factory.initial` and `java.naming.provider.url` JNDI settings, can be configured in the other :) `jndi` element, directly under the `compass` element. Note, the `username` and `password` are used for the `DataSource`, and are completely optional.

4.3.2.5. External

Compass can be configured to use an external data source using the `ExteranlDataSourceProvider`. It uses Java thread local to store the `DataSource` for later use by the data source provider. Setting the data source uses the static method `setDataSource(DataSource dataSource)` on `ExteranlDataSourceProvider`. Here is an example of how it can be configured:

```
<compass name="default">
  <connection>
      <jdbc>
        <dataSourceProvider>
          <external username="testusername" password="testpassword"/>
        </dataSourceProvider>
      </jdbc>
  </connection>
</compass>
```

Note, the `username` and `password` are used for the `DataSource`, and are completely optional.#

4.3.3. File Entry Handler

Configuring the Jdbc store with Compass also allows defining `FileEntryHandler` settings for different file entries in the database. FileEntryHandlers are explained in Appendix B, *Lucene Jdbc Directory* (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 comes free with Compass is automatically using the more performant

`FetchPerTransactoinJdbcIndexInput` 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.

File entries configuration are associated with a 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 characters).

Here is an example of the most common configuration of file entries, changing their buffer size for both index input (used for reading data) and index output (used for writing data):

```
<compass name="default">
  <connection>
      <jdbc>
        <dataSourceProvider>
          <external username="testusername" password="testpassword"/>
        </dataSourceProvider>

        <fileEntries>
            <fileEntry name="__default__">
                <indexInput bufferSize="4096" />
                <indexOutput bufferSize="4096" />
            </fileEntry>
        </fileEntries>
      </jdbc>
  </connection>
</compass>
```

4.3.4. DDL

Compass by default can create the database schema, and has defaults for the column names, types, sizes and so on. The schema definition is configurable as well, here is an example of how to configure it:

```
<compass name="default">
  <connection>
      <jdbc>
        <dataSourceProvider>
          <external username="testusername" password="testpassword"/>
        </dataSourceProvider>

        <ddl>
            <nameColumn name="myname" length="70" />
            <sizeColumn name="mysize" />
        </ddl>
      </jdbc>
  </connection>
</compass>
```

Compass by default will drop the tables when deleting the index, and create them when creating the index. If performing schema based operations is not allowed, the `disableSchemaOperations` flag can be set to `true`. This will cause Compass not to perform any schema based operations.

4.4. Lock Factory

Lucene allows to use different `LockFactory` implementation controlling how locks are performed. By default, each directory comes with its own default lock, but overriding the lock factory can be done within Compass configuration. Here is an example of how this can be done:

```
<compass name="default">
  <connection>
      <file path="target/test-index" />
      <lockFactory type="nativefs" path="test/#subindex#" />
  </connection>
</compass>
```

The lock factory type can have the following values: `simplefs`, `nativefs` (both file system based locks), `nolock`, and `singleinstance`. A fully qualified class name of `LockFactory` implementation or `LockFactoryProvider` can also be provided.

The path allows to provide path parameter to the file system based locks. This is an optional parameter and defaults to the sub index location. The specialized keyword `#subindex#` can be used to be replaced with the actual sub index.

4.5. Local Directory Cache

Compass supports local directory cache implementation allowing to have a local cache per sub index or globally for all sub indexes (that do not have a local cache already specifically defined for them). Local cache can be really useful where a certain sub index is heavily accessed and a local in memory cache is required to improve its performance. Another example is using a local file system based cache when working with a Jdbc directory.

Local Cache fully supports several Compass instances running against the same directory (unlike the directory wrappers explained in the next section) and keeps its local cache state synchronized with external changes periodically.

Here is an example configuring a memory based local cache for sub index called a:

```
<compass name="default">
  <connection>
      <file path="target/test-index" />
      <localCache subIndex="a" connection="ram://" />
  </connection>
</compass>
```

And here is an example of how it can be configured to use local file system cache for all different sub indexes (using the special `__default__` keyword):

```
<compass name="default">
  <connection>
      <file path="target/test-index" />
      <localCache subIndex="__default__" connection="file://tmp/cache" />
  </connection>
</compass>
```

Other than using a faster local cache directory implementation, Compass also improve compound file structure performance by performing the compound operation on the local cache and only flushing the already compound index structure.

4.6. Lucene Directory Wrapper

All the different connection options end up as an instance of a Lucene `Directory` per sub index. Compass provides the ability to wrap the actual `Directory` (think of it as a `Directory` aspect). In order to configure a wrapper, `DirectoryWrapperProvider` implementation must be provided. The `DirectoryWrapperProvider` implementation must implement `Directory wrap(String subIndex, Directory dir)`, which accepts the actual directory and the sub index it is associated with, and return a wrapped `Directory` implementation.

Compass comes with several built in directory wrappers:

4.6.1. SyncMemoryMirrorDirectoryWrapperProvider

Wraps the given Lucene directory with `SyncMemoryMirrorDirectoryWrapper` (which is also provided by Compass). The wrapper wraps the directory with an in memory directory which mirrors it synchronously.

The original directory is read into memory when the wrapper is constructed. All read related operations are performed against the in memory directory. All write related operations are performed both against the in memory directory and the original directory. Locking is performed using the in memory directory.

The wrapper will allow for the performance gains that comes with an in memory index (for read/search operations), while still maintaining a synchronized actual directory which usually uses a more persistent store than memory (i.e. file system).

This wrapper will only work in cases when either the index is read only (i.e. only search operations are performed against it), or when there is a single instance which updates the directory.

Here is an example of how to configure a directory wrapper:

```
<compass name="default">
  <connection>
    <file path="target/test-index"/>
    <directoryWrapperProvider name="test"
      type="org.compass.core.lucene.engine.store.wrapper.SyncMemoryMirrorDirectoryWrapperProvider">
    </directoryWrapperProvider>
  </connection>
</compass>
```

4.6.2. AsyncMemoryMirrorDirectoryWrapperProvider

Wraps the given Lucene directory with `AsyncMemoryMirrorDirectoryWrapper` (which is also provided by Compass). The wrapper wraps the directory with an in memory directory which mirrors it asynchronously.

The original directory is read into memory when the wrapper is constructed. All read related operations are performed against the in memory directory. All write related operations are performed against the in memory directory and are scheduled to be performed against the original directory (in a separate thread). Locking is performed using the in memory directory.

The wrapper will allow for the performance gains that comes with an in memory index (for read/search operations), while still maintaining an asynchronous actual directory which usually uses a more persistent store than memory (i.e. file system).

This wrapper will only work in cases when either the index is read only (i.e. only search operations are performed against it), or when there is a single instance which updates the directory.

Here is an example of how to configure a directory wrapper:

```
<compass name="default">
  <connection>
    <file path="target/test-index"/>
    <directoryWrapperProvider name="test"
      type="org.compass.core.lucene.engine.store.wrapper.AsyncMemoryMirrorDirectoryWrapperProvider">
      <setting name="awaitTermination">10</setting>
      <setting name="sharedThread">true</setting>
    </directoryWrapperProvider>
  </connection>
</compass>
```

`awaitTermination` controls how long the wrapper will wait for the async write tasks to finish. When closing

Compass, there might be still async tasks pending to be written to the actual directory, and the setting control how long (in seconds) Compass will wait for tasks to be executed against the actual directory. `sharedThread` set to `false` controls if each sub index will have its own thread to perform pending "write" operations. If it is set to `true`, a single thread will be shared among all the sub indexes.

Chapter 5. Search Engine

5.1. Introduction

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

5.2. 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 e-mail 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. The alias acts as the connection between a `Resource` and its mapping definitions (OSEM/XSEM/RSEM). 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.

Every `Resource` is associated with one or more id properties. They are required for Compass 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 RSEM definitions or implicitly in OSEM/XSEM definitions.

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

5.2.1. Using Resource/Property

When working with RSEM, resources acts as your prime data model. They are used to construct searchable content, as well as manipulate it. When performing a search, resources be used to display the search results.

Another important place where resources can be used, which is often ignored, is with OSEM/XSEM. When manipulating search content through the use of the application domain model (in case of OSEM), or through the use of xml data structures (in case of XSEM), resources are rarely used. They can be used when performing search operations. Based on your mapping definition, the semantic model could be accessed in a uniformed way through resources and properties.

Lets simplify this statement by using an example. If our application has two object types, Recipe and Ingredient, we can map both recipe title and ingredient title into the same semantic meta-data name, `title` (Resource Property name). This will allow us when searching to display the search results (hits) only on the Resource level, presenting the value of the property title from the list of resources returned.

5.3. 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 usually important to use the same `Analyzer` for both indexing and searching.

`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 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]
```

Lucene also comes with an extension library, holding many more analyzer implementations (including language specific analyzers). Compass can be configured to work with all of them as well.

5.3.1. Configuring Analyzers

A Compass instance acts as a registry of analyzers, with each analyzer bound to a lookup name. Two internal analyzer names within Compass are: `default` and `search`. `default` is the default analyzer that is used when no other analyzer is configured (configuration of using different analyzer is usually done in the mapping definition by referencing a different analyzer lookup name). `search` is the analyzer used on a search query string when no other analyzer is configured (configuring a different analyzer when executing a search based on a query string is done through the query builder API). By default, when nothing is configured, Compass will use Lucene standard analyzer as the `default` analyzer.

The following is an example of configuring two analyzers, one that will replace the `default` analyzer, and another one registered against `myAnalyzer` (it will probably later be referenced from within the different mapping definitions).

```
<compass name="default">

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

    <searchEngine>
        <analyzer name="deault" type="Snowball" snowballType="Lovins">
            <stopWords>
                <stopWord value="no" />
            </stopWords>
        </analyzer>
        <analyzer name="myAnalyzer" type="Standard" />
    </searchEngine>
</compass>
```

Compass also supports custom implementations of Lucene `Analyzer` class (note, the same goal might be achieved by implementing an analyzer filter, described later). If the implementation also implements `CompassConfigurable`, additional settings (parameters) can be injected to it using the configuration file. Here is an example configuration that registers a custom analyzer implementation that accepts a parameter named threshold:

```
<compass name="default">

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

    <searchEngine>
```

```
        <analyzer name="deault" type="CustomAnalyzer" analyzerClass="eg.MyAnalyzer">
          <setting name="threshold">5</setting>
        </analyzer>
    </searchEngine>
</compass>
```

5.3.2. 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.

A custom filter implementation need to implement Compass `LuceneAnalyzerTokenFilterProvider`, which single method creates a Lucene `TokenFilter`. Filters are registered against a name as well, which can then be used in the analyzer configuration to reference them. The next example configured two analyzer filters, which are applied on to the `default` analyzer:

```
<compass name="default">

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

  <searchEngine>
      <analyzer name="deafult" type="Standard" filters="test1, test2" />

      <analyzerFilter name="test1" type="eg.AnalyzerTokenFilterProvider1">
          <setting name="param1" value="value1" />
      </analyzerFilter>
      <analyzerFilter name="test2" type="eg.AnalyzerTokenFilterProvider2">
          <setting name="paramX" value="valueY" />
      </analyzerFilter>
  </searchEngine>
</compass>
```

5.3.3. Handling Synonyms

Since synonyms are a common requirement with a search application, Compass comes with a simple synonym analyzer filter: `SynonymAnalyzerTokenFilterProvider`. The implementation requires as a parameter (setting) an implementation of a `SynonymLookupProvider`, which can return all the synonyms for a given value. No implementation is provided, though one that goes to a public synonym database, or a file input structure is simple to implement. Here is an example of how to configure it:

```
<compass name="default">

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

  <searchEngine>
      <analyzer name="deafult" type="Standard" filters="synonymFilter" />

      <analyzerFilter name="synonymFilter" type="synonym">
          <setting name="lookup" value="eg.MySynonymLookupProvider" />
      </analyzerFilter>
  </searchEngine>
</compass>
```

Note the fact that we did not set the fully qualified class name for the type, and used `synonym`. This is a simplification that comes with Compass (naturally, you can still use the fully qualified class name of the synonym token filter provider).

5.4. Query Parser

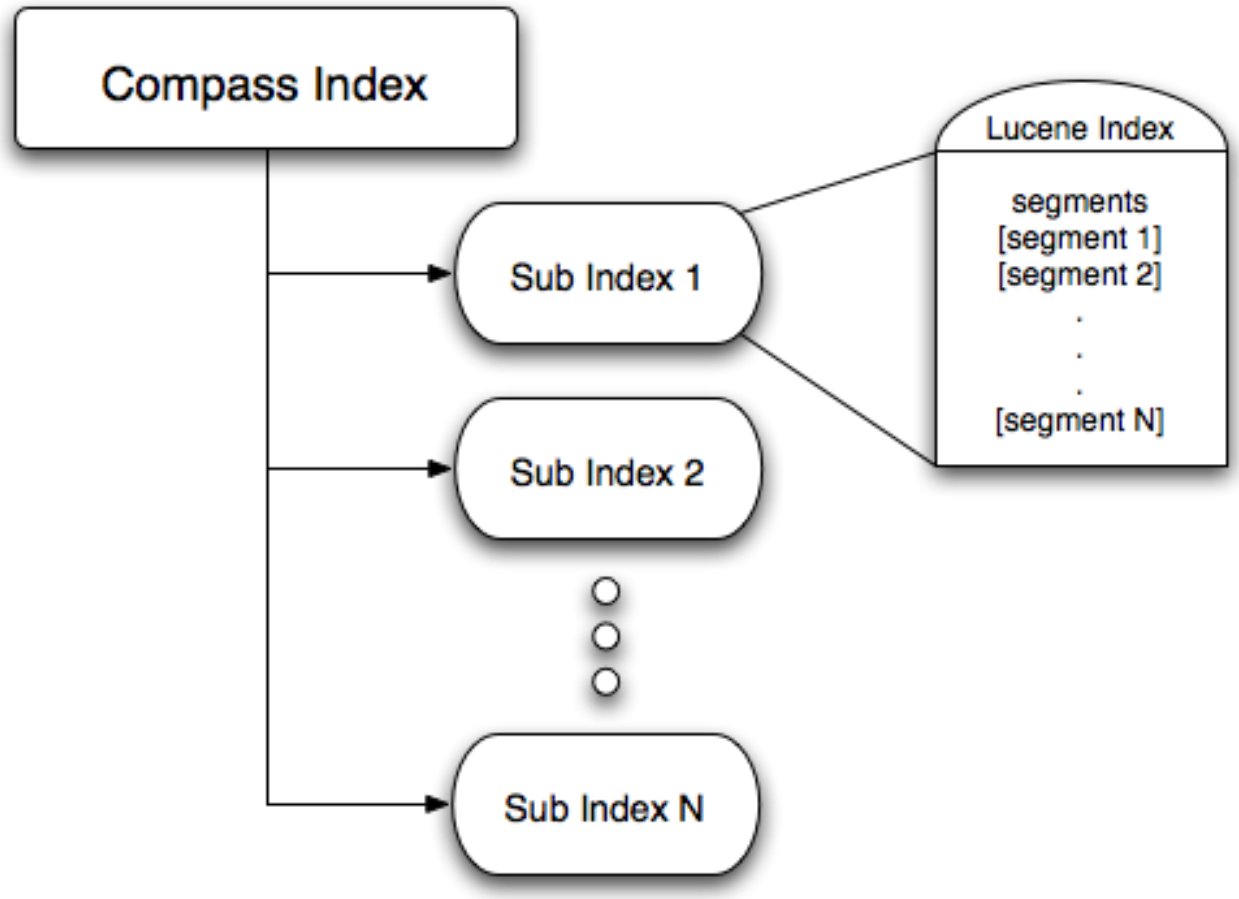
By default, Compass uses its own query parser based on Lucene query parser. Compass allows to configure several query parsers (registered under a lookup name), as well as override the default Compass query parser (registered under the name `default`). Custom query parsers can be used to extend the default query language support, to add parsed query caching, and so on. A custom query parser must implement the `LuceneQueryParser` interface.

Here is an example of configuring a custom query parser registered under the name `test`:

```
<compass name="default">

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

  <searchEngine>
    <queryParser name="test" type="eg.MyQueryParser">
      <setting name="param1" value="value1" />
    </queryParser>
  </searchEngine>
</compass>
```

5.5. Index Structure

It is very important to understand how the Search Engine index is organized so we can than talk about transaction, optimizers, and sub index hashing. The following structure shows the Search Engine Index Structure:



Compass Index Structure

Every sub-index has it's own fully functional index structure (which maps to a single Lucene index). The Lucene index part holds a "meta data" file about the index (called `segments`) and 0 to N segment files. The segments can be a single file (if the compound setting is enabled) or multiple files (if the compound setting is disable). A segment is close to a fully functional index, which hold the actual inverted index data (see Lucene documentation for a detailed description of these concepts).

Index partitioning is one of Compass main features, allowing for flexible and configurable way to manage complex indexes and performance considerations. The next sections will explain in more details why this feature is important, especially in terms of transaction management.

5.6. Transaction

Compass Search Engine abstraction provides support for transaction management on top of Lucene. The abstraction support common transaction levels: `read_committed` and `serializable`, as well as the special `batch_insert` one. Compass provides two phase commit support for the common transaction levels only.

5.6.1. Locking

Compass utilizes Lucene inter and outer process locking mechanism and uses them to establish it's transaction locking. Note that the transaction locking is on the "sub-index" level (the sub index based index), which means that dirty operations only lock their respective sub-index index. So, the more aliases / searchable content map to the same index (next section will explain how to do it - called sub index hashing), the more aliases / searchable content will be locked when performing dirty operations, yet the faster the searches will be. Lucene uses a

special lock file to manage the inter and outer process locking which can be set in the Compass configuration. You can manage the transaction timeout and polling interval using Compass configuration.

A Compass transaction acquires a lock only when a dirty (i.e. `create`, `save` or `delete`) operation occurs, which makes "read only" transactions as fast as they should and can be. The following configuration file shows how to control the two main settings for locking, the locking timeout (which defaults to 10 seconds) and the locking polling interval (how often Compass will check and see if a lock is released or not) (defaults to 100 milli-seconds):

```
<compass name="default">

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

  <transaction lockTimeout="15" lockPollInterval="200" />
</compass>
```

5.6.2. Isolation

5.6.2.1. read_committed

Read committed transaction isolation level allows to isolate changes done during a transaction from other transactions until commit. It also allows for load/get/find operations to take into account changes done during the current transaction. This means that a delete that occurs during a transaction will be filtered out if a search is executed within the same transaction just after the delete.

When starting a `read_committed` transaction, no locks are obtained. Read operation will not obtain a lock either. A lock will be obtained only when a dirty operation is performed. The lock is obtained only on the index of the alias / searchable content that is associated with the dirty operation, i.e the sub-index, and will lock all other aliases / searchable content that map to that sub-index. In Compass, 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, fast updates, as well as paving the way for a two phase commit support (and the reason behind having optimizers).

The read committed transaction support concurrent commit where if operations are performed against several sub indexes, the commit process will happen concurrently on the different sub indexes. It uses Compass internal Execution Manager where the number of threads as well as the type of the execution manager (concurrent or work manager) can be configured.

5.6.2.2. 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).

5.6.2.3. lucene

A special transaction level, `lucene` (previously known as `batch_insert`) isolation level is similar to the `read_committed` isolation level except dirty operations done during a transaction are not visible to get/load/find operations that occur within the same transaction. This isolation level is very handy for long running batch dirty operations and can be faster than `read_committed`. Most usage patterns of Compass (such as integration with ORM tools) can work perfectly well with the `lucene` isolation level.

It is important to understand this transaction isolation level in terms of merging done during commit time. Lucene might perform some merges during commit time depending on the merge factor configured using `compass.engine.mergeFactor`. This is different from the `read_committed` isolation level where no merges are perfomed during commit time. Possible merges can cause commits to take some time, so one option is to configure a large merge factor and let the optimizer do its magic (you can configure a different merge factor for the optimizer).

Another important parameter when using this transaction isolation level is `compass.engine.ramBufferSize` (defaults to 16.0 Mb) which replaces the max buffered docs parameter and controls the amount of transactional data stored in memory. Larger values will yield better performance and it is best to allocate as much as possible.

Most of the parameters can also be configured on a per session/transaction level. Please refer to `RuntimeLuceneEnvironment` for more information.

The lucene transaction support concurrent commit where if operations are performed against several sub indexes, the commit process will happen concurrently on the different sub indexes. It uses Compass internal Execution Manager where the number of threads as well as the type of the execution manager (concurrent or work manager) can be configured.

Here is how the transaction isolation level can be configured:

```
<compass name="default">
  <connection>
      <file path="target/test-index" />
  </connection>
  <transaction isolation="lucene" />
</compass>
```

```
compass.engine.connection=target/test-index
compass.transaction.isolation=lucene
```

5.6.3. Transaction Log

For `read_committed` and `serializable` transaction isolation Compass uses a transaction log of the current transaction data running. Compass provides the following transaction log implementations:

5.6.3.1. Ram Transaction Log

The Ram transaction log stores all the transaction information in memory. This is the fastest transaction log available and is the default one Compass uses. The transaction size is controlled by the amount of memory the JVM has.

Even though this is the default transaction log implementation, here is how it can be configured:

```
<compass name="default">
  <connection>
      <file path="target/test-index" />
  </connection>
  <transaction isolation="read_committed">
    <readCommittedSettings transLog="ram://" />
  </transaction>
</compass>
```

```
compass.engine.connection=target/test-index
compass.transaction.readcommitted.translog.connection=ram://
```

5.6.3.2. FS Transaction Log

The FS transaction log stores the transactional data on the file system. This allows for bigger transactions (bigger in terms of data) to be run when compared with the ram transaction log though on account of performance. The fs transaction log can be configured with a path where to store the transaction log (defaults to java.io.tmpdir system property). The path is then appended with `compass/translog` and for each transaction a new unique directory is created.

Here is an example of how the fs transaction can be configured:

```
<compass name="default">
<connection>
    <file path="target/test-index" />
</connection>
<transaction isolation="read_committed">
  <readCommittedSettings transLog="file://" />
</transaction>
</compass>
```

```
compass.engine.connection=target/test-index
compass.transaction.readcommitted.translog.connection=file://
```

Transactional log settings are one of the session level settings that can be set. This allows to change how Compass would save the transaction log per session, and not globally on the Compass instance level configuration. Note, this only applies on the session that is responsible for creating the transaction. The following is an example of how it can be done:

```
CompassSession session = compass.openSession();
session.getSettings().setSetting(RuntimeLuceneEnvironment.Transaction.ReadCommittedTransLog.CONNECTION,
                                 "file://tmp/");
```

5.7. All Support

When indexing an Object, XML, or a plain Resource, their respective properties are added to the index. These properties can later be searched explicitly, for example: `title:fang`. Most times users wish to search on all the different properties. For this reason, Compass, by default, supports the notion of an "all" property. The property is actually a combination of the different properties mapped to the search engine.

The all property provides advance features such using declared mappings of given properties. For example, if a property is marked with a certain analyzer, that analyzer will be usde to add the property to the all property. If it is untokenized, it will be added without analyzing it. If it is configured with a certain boost value, that part of the all property, when "hit", will result in higher ranking of the result.

The all property allows for global configuration and per mapping configuration. The global configuration allows to disable the all feature completely (`compass.property.all.enabled=false`). It allows to exclude the alias from the all proeprty (`compass.property.all.excludeAlias=true`), and can set the term vector for the all property (`compass.property.all.termVector=yes` for example).

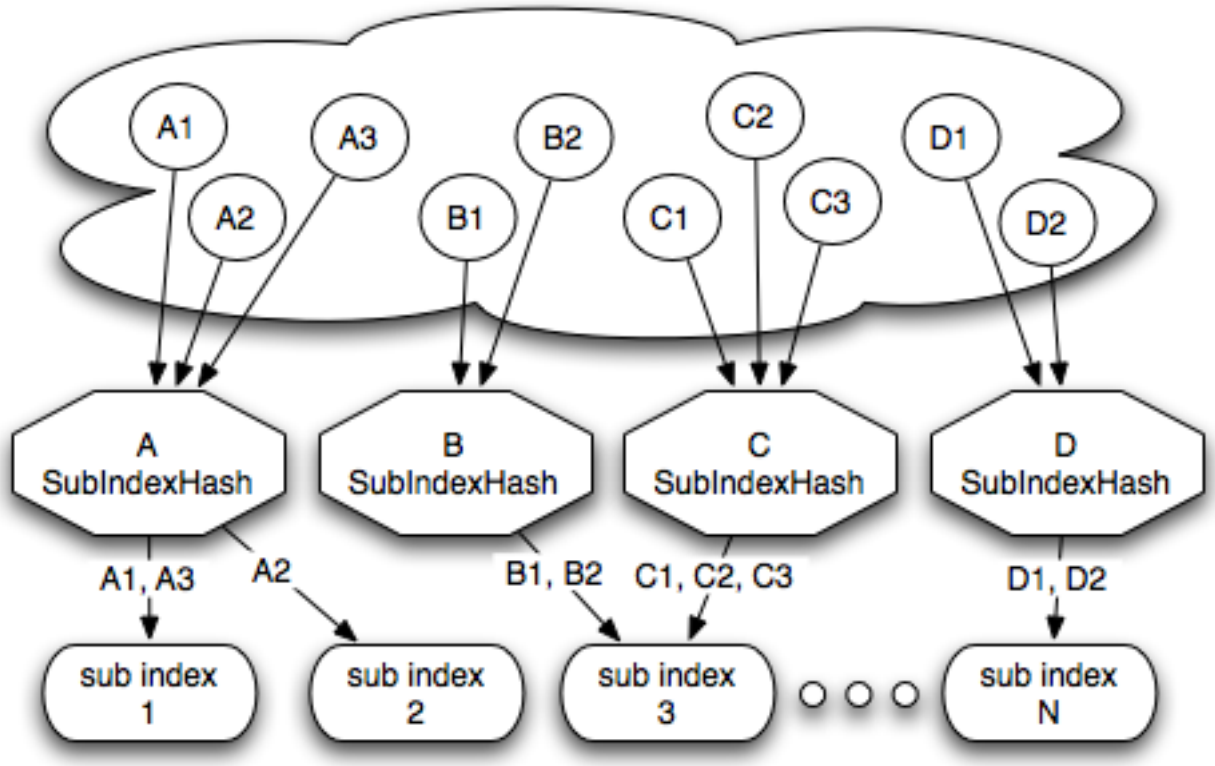
The per mapping definitions allow to configure the above settings on a mapping level (they override the global ones). They are included in an `all` tag that should be the first one within the different mappings. Here is an example for OSEM:

```
<compass-core-mapping>
<[mapping] alias="test-alias">
  <all enable="true" exclude-alias="true" term-vector="yes" omit-norms="yes" />
</[mapping]>
```

```
</compass-core-mapping>
```

5.8. Sub Index Hashing

Searchable content is mapped to the search engine using Compass different mapping definitions (OSEM/XSEM/RSEM). Compass provides the ability to partition the searchable content into different sub indexes, as shown in the next diagram:



Sub Index Hashing

In the above diagram A, B, C, and D represent aliases which in turn stands for the mapping definitions of the searchable content. A1, B2, and so on, are actual instances of the mentioned searchable content. The diagram shows the different options of mapping searchable content into different sub indexes.

5.8.1. Constant Sub Index Hashing

The simplest way to map aliases (stands for the mapping definitions of a searchable content) is by mapping all its searchable content instances into the same sub index. Defining how searchable content mapping to the search engine (OSEM/XSEM/RSEM) is done within the respectable mapping definitions. There are two ways to define a constant mapping to a sub index, the first one (which is simpler) is:

```
<compass-core-mapping>
  <[mapping] alias="test-alias" sub-index="test-subindex">
    <!-- ... -->
  </[mapping]>
</compass-core-mapping>
```

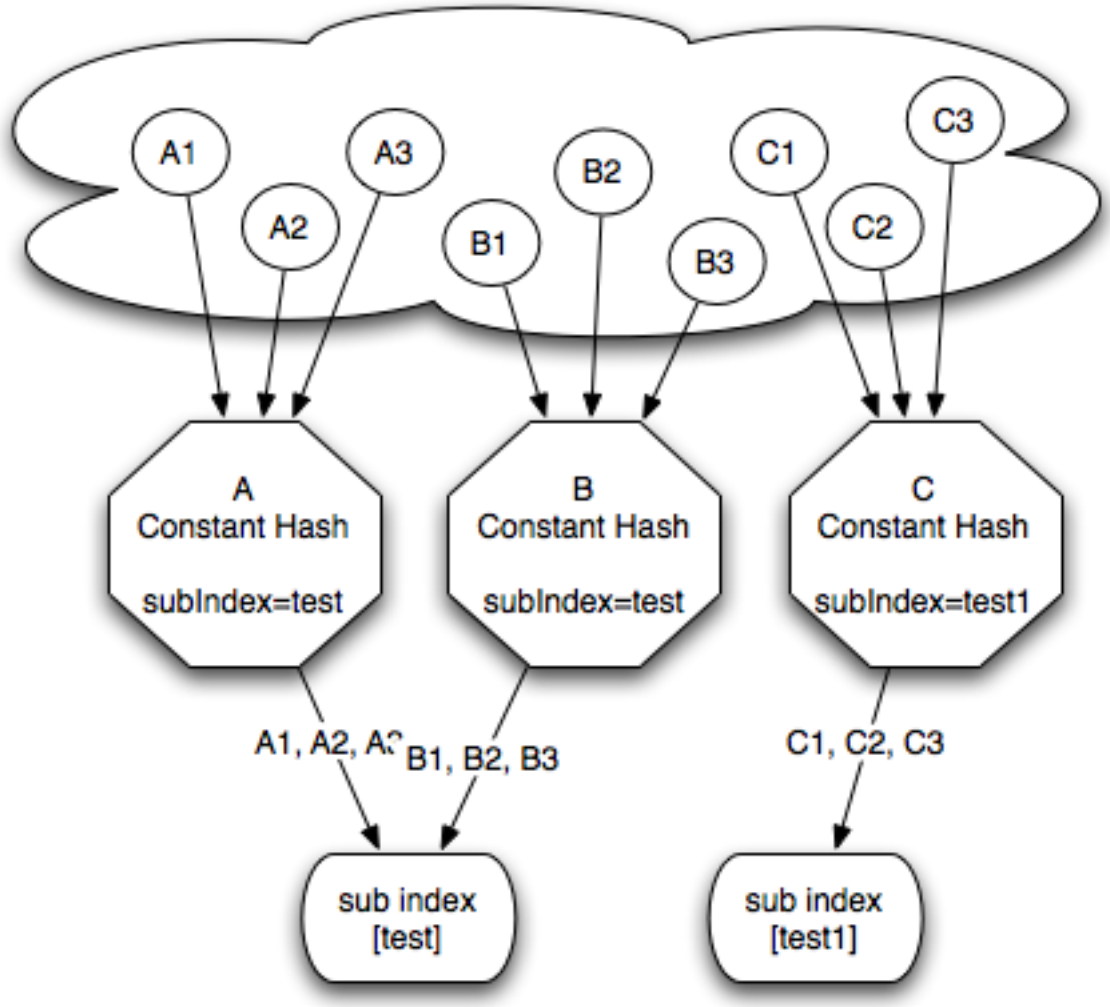
The mentioned `[mapping]` that is represented by the alias `test-alias` will map all its instances to

`test-subindex`. Note, if `sub-index` is not defined, it will default to the `alias` value.

Another option, which probably will not be used to define constant sub index hashing, but shown here for completeness, is by specifying the constant implementation of `SubIndexHash` within the mapping definition (explained in details later in this section):

```
<compass-core-mapping>
  <[mapping] alias="test-alias">
    <sub-index-hash type="org.compass.core.engine.subindex.ConstantSubIndexHash">
        <setting name="subIndex" value="test-subindex" />
    </sub-index-hash>
    <!-- ... -->
  </[mapping]>
</compass-core-mapping>
```

Here is an example of how three different aliases: A, B and C can be mapped using constant sub index hashing:

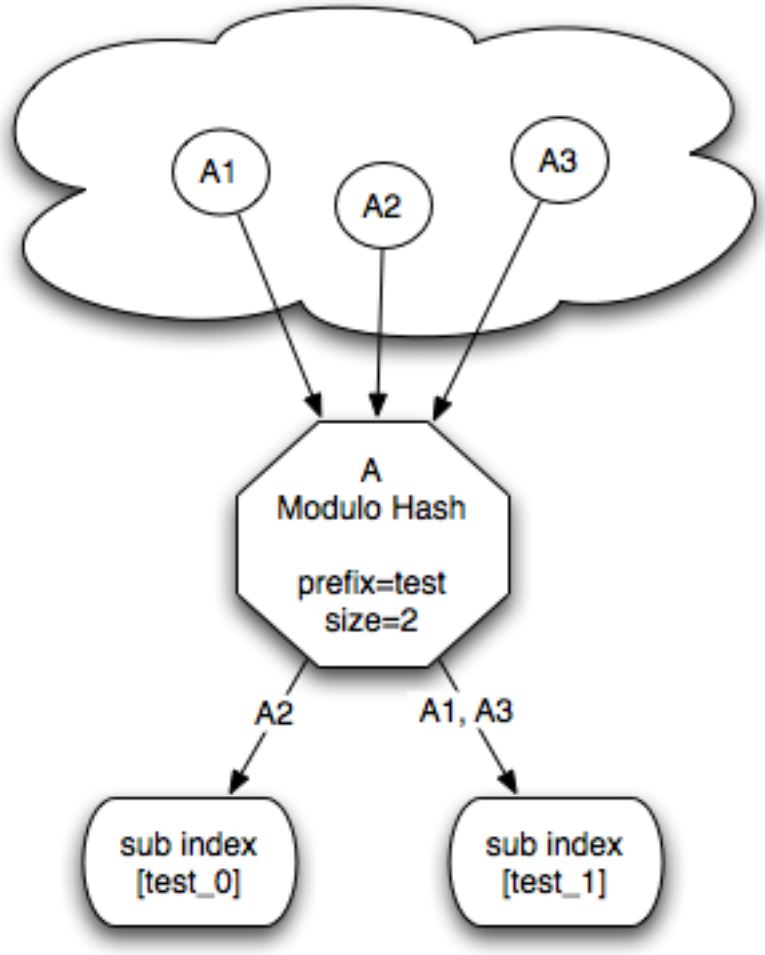


Modulo Sub Index Hashing

5.8.2. Modulo Sub Index Hashing

Constant sub index hashing allows to map an alias (and all its searchable instances it represents) into the same sub index. The modulo sub index hashing allows for partitioning an alias into several sub indexes. The partitioning is done by hashing the alias value with all the string values of the searchable content ids, and then using the modulo operation against a specified size. It also allows setting a constant prefix for the generated sub

index value. This is shown in the following diagram:

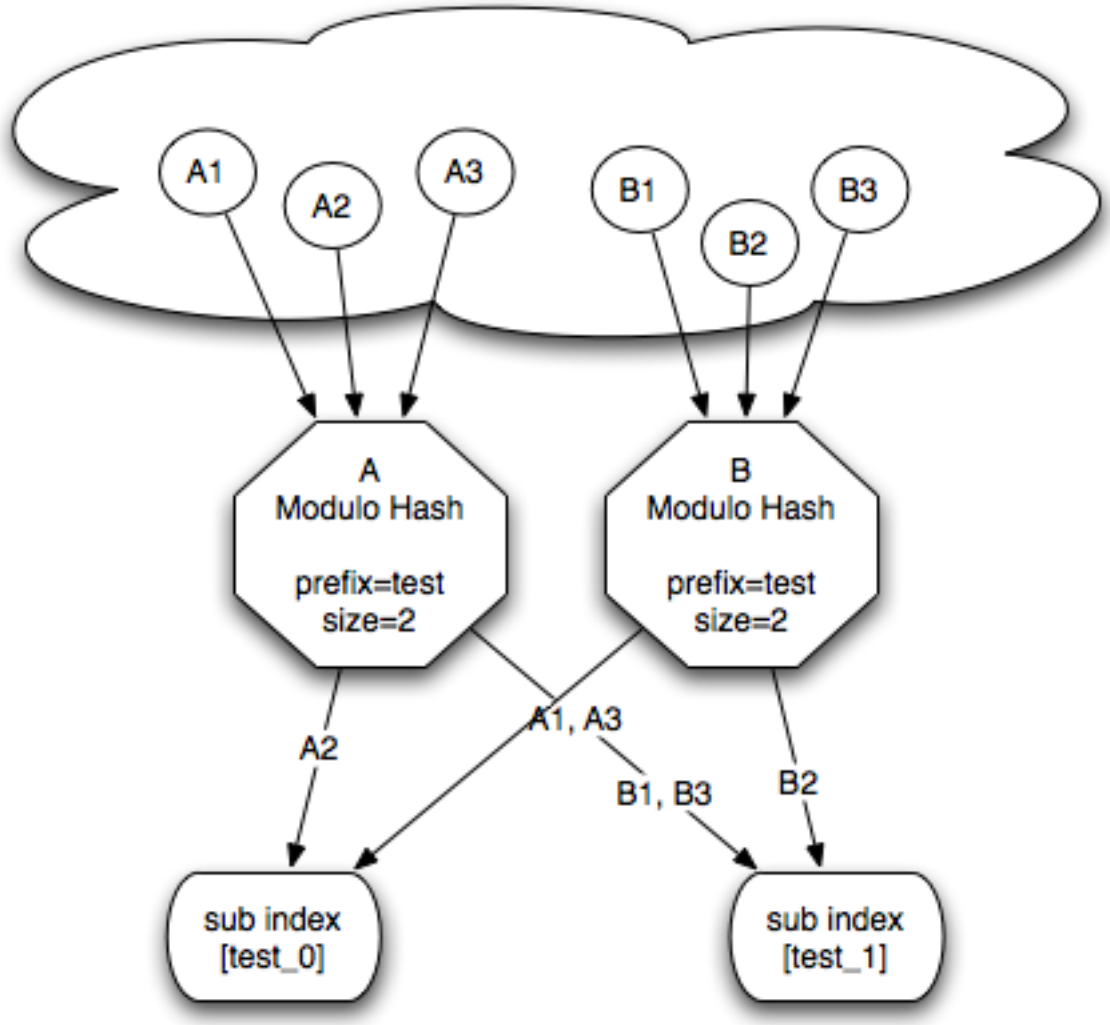


Modulo Sub Index Hashing

Here, A1, A2 and A3 represent different instances of alias A (let it be a mapped Java class in OSEM, a Resource in RSEM, or an XmlObject in XSEM), with a single id mapping with the value of 1, 2, and 3. A modulo hashing is configured with a prefix of `test`, and a size of 2. This resulted in the creation of 2 sub indexes, called `test_0` and `test_1`. Based on the hashing function (the alias String hash code and the different ids string hash code), instances of A will be directed to their respective sub index. Here is how A alias would be configured:

```
<compass-core-mapping>
  <[mapping] alias="A">
    <sub-index-hash type="org.compass.core.engine.subindex.ModuloSubIndexHash">
        <setting name="prefix" value="test" />
        <setting name="size" value="2" />
    </sub-index-hash>
    <!-- ... -->
  </[mapping]>
</compass-core-mapping>
```

Naturally, more than one mapping definition can map to the same sub indexes using the same modulo configuration:



Complex Modulo Sub Index Hashing

5.8.3. Custom Sub Index Hashing

`ConstantSubIndexHash` and `ModuloSubIndexHash` are implementation of Compass `SubIndexHash` interface that comes built in with Compass. Naturally, a custom implementation of the `SubIndexHash` interface can be configured in the mapping definition.

An implementation of `SubIndexHash` must provide two operations. The first, `getSubIndexes`, must return all the possible sub indexes the sub index hash implementation can produce. The second, `mapSubIndex(String alias, Property[] ids)` uses the provided aliases and ids in order to compute the given sub index. If the sub index hash implementation also implements the `CompassConfigurable` interface, different settings can be injected to it. Here is an example of a mapping definition with custom sub index hash implementation:

```
<compass-core-mapping>
  <[mapping] alias="A">
    <sub-index-hash type="eg.MySubIndexHash">
        <setting name="param1" value="value1" />
        <setting name="param2" value="value2" />
    </sub-index-hash>
    <!-- ... -->
  </[mapping]>
</compass-core-mapping>
```

5.9. Optimizers

As mentioned in the `read_committed` section, every dirty transaction that is committed successfully creates another segment in the respective sub 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 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 `Compass` is closed, the `SearchEngineOptimizer` is stopped.

The optimization process works on a sub index level, performing the optimization for each one. During the optimization process, optimizers will lock the sub index for dirty operations. This causes a tradeoff between having an optimized index, and spending less time on the optimization process in order to allow for other dirty operations.

5.9.1. Scheduled Optimizers

Each optimizer in Compass can be wrapped to be executed in a scheduled manner. The default behavior within Compass is to schedule the configured optimizer (unless it is the null optimizer). Here is a sample configuration file that controls the scheduling of an optimizer:

```
<compass name="default">

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

  <searchEngine>
    <optimizer scheduleInterval="90" schedule="true" />
  </searchEngine>
</compass>
```

5.9.2. 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).

Since this optimizer causes all the segments in the index to be optimized into a single segment, the optimization process might take a long time to happen. This means that for large indexes, the optimizer will block other dirty operations for a long time in order to perform the index optimization. It also means that the index will be fully optimized after it, which means that search operations will execute faster. For most cases, the `AdaptiveOptimizer` should be the one used.

5.9.3. Adaptive Optimizer

The `AdaptiveOptimizer` optimizes the segments while trying to keep 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 then 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).

5.9.4. Null Optimizer

Compass 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. It can also be used when the index was built offline and has been fully optimized, and later it is only used for search/read operations.

5.10. Merge

Lucene perfoms merges of different segments after certain operaitons are done on the index. The less merges you have, the faster the searching is. The more merges you do, the slower certain operations will be. Compass allows for fine control over when merges will occur. This depends greatly on the transaction isolation level and the optimizer used and how they are configured.

5.10.1. Merge Policy

Merge policy controls which merges are supposed to happen for a ceratin index. Compass allows to simply configure the two merge policies that come with Lucene, the `LogByteSize` (the default) and `LogDoc`, as well as configure custom implementations. Configuring the type can be done usign `compass.engine.merge.policy.type` and has possible values of `logbytesize`, `logdoc`, or the fully qualified class name of a `MergePolicyProvider`.

The `LogByteSize` can be further configured using `compass.engine.merge.policy.maxMergeMB` and `compass.engine.merge.policy.minMergeMB`.

5.10.2. Merge Scheduler

Merge scheduler controls how merge operations happen once a merge is needed. Lucene comes with built in `ConcurrentMergeSchduler` (executes merges concurrently on newly created threads) and `SerialMergeScheduler` that executes the merge operations on the same therad. Compass extends Lucene and provide `ExecutorMergeScheduler` allowing to utlize Compass internal exdecutor pool (either concurrent or work manager backed) with no overhead of creating new threads. This is the default merge scheduler that comes with Compass.

Configuring the type of the merge scheduler can be done using `compass.engine.merge.scheduler.type` with the following possible values: `executor` (the default), `concurrent` (Lucene Concurrent merge scheduler), and `serial` (Lucene serial merge scheduler). It can also have a fully qualified name of an implementation of `MergeSchedulerProvider`.

5.11. Index Deletion Policy

Lucene allows to define an `IndexDeletionPolicy` which allows to control when commit points are deleted from the index storage. Index deletion policy mainly aim at allowing to keep old Lucene commit points relevant for a certain parameter (such as expiration time or number of commits), which allows for better NFS support for example. Compass allows to easily control the index deletion policy to use and comes built in with several index deletion policy implementations. Here is an example of its configuration using the default index deletion policy which keeps only the last commit point:

```
<compass name="default">
```

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

  <searchEngine>
    <indexDeletionPolicy>
        <keepLastCommit />
    </indexDeletionPolicy>
  </searchEngine>
</compass>
```

Here is the same configuration using properties based configuration:

```
<compass name="default">

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

  <settings>
      <setting name="compass.engine.store.indexDeletionPolicy.type" value="keeplastcommit" />
  </settings>
</compass>
```

Compass comes built in with several additional deletion policies including: `keepall` which keeps all commit points. `keeplastn` which keeps the last N commit points. `expirationtime` which keeps commit points for X number of seconds (with a default expiration time of "cache invalidation interval * 3").

By default, the index deletion policy is controlled by the actual index storage. For most (ram, file) the deletion policy is keep last committed (which should be changed when working over a shared disk). For distributed ones (such as coherence, gigaspaces, terrracotta), the index deletion policy is the expiration time one.

5.12. Spell Check / Did You Mean

Compass comes with built in support for spell check support. It allows to suggest queries (did you mean feature) as well as allow to get possible suggestions for given words. By default, the spell check support is disabled. In other to enable it, the following property need to be set:

```
compass.engine.spellcheck.enable=true
```

Once spell check is enabled, a special spell check index will be built based on the "all" property (more on that later). It can then be used in the following simple manner:

```
CompassQuery query = session.queryBuilder().queryString("jack london").toQuery();
CompassHits hits = query.hits();
System.out.println("Original Query: " + hits.getQuery());
if (hits.getSuggestedQuery().isSuggested()) {
    System.out.println("Did You Mean: " + hits.getSuggestedQuery());
}
```

In order to perform spell index level operations, `Compass` exposes now a `getSpellCheckManager()` in order to perform them. Note, this method will return `null` in case spell check is disabled. The spell check manager also allows to get suggestions for a given word.

By default, when the spell check index is enabled, two scheduled tasks will kick in. The first scheduled task is responsible for monitoring the spell check index, and if changed (for example, by a different Compass instance), will reload the latest changes into the index. The interval for this scheduled task can be controlled

using the setting `compass.engine.cacheIntervalInvalidation` (which is used by Compass for the actual index as well), and defaults to 5 seconds (it is set in milliseconds).

The second scheduler is responsible for identifying that the actual index was changed, and rebuild the spell check index for the relevant sub indexes that were changed. It is important to understand that the spell check index will not be updated when operations are performed against the actual index. It will only be updated if explicitly called for rebuild or concurrentRebuild using the Spell Check Manager, or through the scheduler (which calls the same methods). By default, the scheduler will run every 10 minutes (no sense in rebuilding the spell check index very often), and can be controlled using the following setting: `compass.engine.spellcheck.scheduleInterval` (resolution in seconds).

5.12.1. Spell Index

Compass by default will build a spell index using the same configured index storage simply under a different "sub context" name called `spellcheck` (the compass index is built under sub context `index`). For each sub index in Compass, a spell check sub index will be created. By default, a scheduler will kick in (by default each 10 minutes) and will check if the spell index needs to be rebuilt, and if it does, it will rebuild it. The spell check manager also exposes API in order to perform the rebuild operations as well as checking if the spell index needs to be rebuilt. Here is an example of how the scheduler can be configured:

```
compass.engine.spellcheck.enable=true
# the default it true, just showing the setting
compass.engine.spellcheck.schedule=true
# the schedule, in minutes (defaults to 10)
compass.engine.spellcheck.scheduleInterval=10
```

The spell check index can be configured to be stored on a different location than the Compass index. Any index related parameters can be set as well. Here is an example (for example, if the index is stored in the database, and spell index should be stored on the file system):

```
compass.engine.spellcheck.enable=true
compass.engine.spellcheck.engine.connection=file://target/spellindex
compass.engine.spellcheck.engine.ramBufferSize=40
```

In the above example we also configure the indexing process of the spell check index to use more memory (40) so the indexing process will be faster. As seen here, settings that control the index can be used (`compass.engine.` settings) can apply to the spell check index by prepending the `compass.engine.spellcheck` setting.

So, what is actually being included in the spell check index. Out of the box, by just enabling spell check, the all field is going to be used to get the terms for the spell check index. In this case, things that are excluded from the all field will be excluded from the spell check index as well

Compass allows for great flexibility in what is going to be included or excluded in the spell check index. The first two important settings are: `compass.engine.spellcheck.defaultMode` and the `spell-check` resource mapping level definition (for class/resource/xml-object). By default, both are set to `NA`, which results in including the all property. The all property can be excluded by setting the `spell-check` to `exclude` on the all mapping definition.

Each resource mapping (resource/class/xml-object) can have a `spell-check` definition of `include`, `exclude`, and `na`. If set to `na`, the global default mode will be used for it (which can be set to `include`, `exclude` and `na` as well).

When the resource mapping ends up with `spell-check` of `include`, it will automatically include all the

properties for the given mapping, except for the "all" property. Properties can be excluded by specifically setting their respective `spell-check` to `exclude`.

When the resource mapping ends up with `spell-check` of `exclude`, it will automatically exclude all the properties for the given mapping, as well as the "all" property. Properties can be included by specifically setting their respective `spell-check` to `include`.

On top of specific mapping definition. Compass can be configured with `compass.engine.spellcheck.globablIncludeProperties` which is a comma separated list of properties that will always be included. And `compass.engine.spellcheck.globablExcludeProperties` which is a comma separated list of properties that will always be excluded.

If you wish to know which properties end up being included for certain sub index, turn the debug logging level on for `org.compass.core.lucene.engine.spellcheck.DefaultLuceneSpellCheckManager` and it will print out the list of properties that will be used for each sub index.

5.13. Direct Lucene

Compass provides a helpful abstraction layer on top of Lucene, but it also acknowledges that there are cases where direct Lucene access, both in terms of API and constructs, is required. Most of the direct Lucene access is done using the `LuceneHelper` class. The next sections will describe its main features, for a complete list, please consult its javadoc.

5.13.1. Wrappers

Compass wraps some of Lucene classes, like `Query` and `Filter`. There are cases where a Compass wrapper will need to be created out of an actual Lucene class, or an actual Lucene class need to be accessed out of a wrapper.

Here is an example for wrapping the a custom implementation of a Lucene `Query` with a `CompassQuery`:

```
CompassSession session = // obtain a compass session
Query myQ = new MyQuery(param1, param2);
CompassQuery myCQ = LuceneHelper.createCompassQuery(session, myQ);
CompassHits hits = myCQ.hits();
```

The next sample shows how to get Lucene `Explanation`, which is useful to understand how a query works and executes:

```
CompassSession session = // obtain a compass session
CompassHits hits = session.find("london");
for (int i = 0; i < hits.length(); i++) {
  Explanation exp = LuceneHelper.getLuceneSearchEngineHits(hits).explain(i);
  System.out.println(exp.toString());
}
```

5.13.2. Searcher And IndexReader

When performing read operations against the index, most of the time Compass abstraction layer is enough. Sometimes, direct access to Lucene own `IndexReader` and `Searcher` are required. Here is an example of using the reader to get all the available terms for the category property name (Note, this is a prime candidate for future inclusion as part of Compass API):

```
CompassSession session = // obtain a compass session
LuceneSearchEngineInternalSearch internalSearch = LuceneHelper.getLuceneInternalSearch(session);
TermEnum termEnum = internalSearch.getReader().terms(new Term("category", ""));
```

```
try {
  ArrayList tempList = new ArrayList();
  while ("category".equals(termEnum.term().field())) {
    tempList.add(termEnum.term().text());

    if (!termEnum.next()) {
        break;
    }
  }
} finally {
  termEnum.close();
}
```

Chapter 6. OSEM - Object/Search Engine Mapping

6.1. Introduction

Compass provides the ability to map Java Objects to the underlying Search Engine using Java 5 Annotations or 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/annotations 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.

The process of saving an Object into the search engine is called marshaling, and the process of retrieving an object from the search engine is called un-marshaling. As described in Section 5.2, “Alias, Resource and Property”, Compass uses Resources when working against a search engine, and OSEM is the process of marshaling and un-marshaling an Object tree to a Resource (for simplicity, think of a Resource as a Map).

6.2. 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;

@Searchable
@SearchableConstant(name = "type", values = {"person", "author"})
public class Author {
  private Long id;  // identifier
  private String name;
  private Date birthday;

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

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

  @SearchableProperty(name = "name")
  @SearchableMetaData(name = "authorName")
  public String getName() {
    return this.name;
  }

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

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

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

The Author class is mapped using Java 5 annotations. The following shows how to map the same class using

OSEM xml mapping definitions:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
    "-//Compass/Compass Core Mapping DTD 2.0//EN"
    "http://www.compass-project.org/dtd/compass-core-mapping-2.0.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 format="yyyy-MM-dd">birthday</meta-data>
    </property>

  </class>
</compass-core-mapping>
```

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

- 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()`
- Provide Property Identifier(s): OSEM requires that a *root searchable* Object will define one or more properties (JavaBean properties) that identifies the class.
- 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 recognizes JavaBean style property (`getFoo`, `isFoo`, and `setFoo`). This mechanism works with any level of visibility.
- It is recommended 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 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..

The above example defines the mapping for `Author` class. 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".

6.2.1. Alias

Each mapping definition in Compass is registered under an alias. The alias is used as the link between the OSEM definitions of a class, and the class itself. The alias can then be used to reference the mapping, both in other mapping definitions and when working directly with Compass API. When using annotations mappings, the alias defaults to the short class name.

6.2.2. Root

There are two types of searchable classes in Compass, root searchable classes and non-root searchable classes. Root searchable classes are best defined as classes that return as hits when a search is performed. For example, in a scenario where we have Customer class with a Name class, the Customer will be a root searchable class, and Name would have root="false" in it since it does not "stands on its own". Another way of looking at root searchable classes is as searchable classes that end up marshaled into their own Resource (which is then used to work against the search engine).

Non root searchable classes are not required to define id mappings.

6.2.3. Sub Index

By default, each *root* searchable class will have its own sub index defaulting to the alias name. The sub index name can be controlled, allowing to join several root searchable classes into the same sub index, or using different sub index hashing functions. Please read Section 5.8, “Sub Index Hashing” for more information.

6.3. Searchable Class Mappings

6.3.1. Searchable Id and Searchable Meta Data

Each *root* searchable class must define at least one searchable id. The searchable id(s) are used to uniquely identify the object within its alias context. More than one searchable id can be defined, as well as user defined classes to act as searchable ids (must register its own converter or use searchable id component mapping).

Searchable Id does not require the definition of a searchable meta-data. If none is defined, Compass will automatically create an internal meta-data id (explained later) which most times is perfectly fine (usually, text searching based on the surrogate id is not required). If the searchable id does need to be searched, a searchable meta-data need to be defined for it. When using xml mapping, one or more meta-data element need to be added to the id element. When using annotations, there are three options: the first, provide a name for the SearchableId (compass will automatically act as if a SearchableMetaData was defined on the SearchableId and add it), the second is to add a SearchableMetaData annotation and the last is to add SearchableMetaDatas annotation (for multiple meta-datas). Of course, all the three can be combined. The reason why SearchalbeId will automatically create a SearchableMetaData if the name is provided is to ease the number of annotations required (and not get to annotation hell).

Here is an example of defining a Searchable Id using annotations. This example will not create any visible meta-data (as the SearchableId has no name to it, or SearchableMetaData(s) annotation).

```
@Searchable
public class Author {
  @SearchableId
  private Long id;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="Author" alias="author">
  <id name="id" />
  <!-- ... -->
</class>
```

The following is another example, now with actually defining a meta-data on the id for its values to be searchable:

```
@Searchable
public class Author {
  @SearchableId(name = "id")
  private Long id;
  // ...
}
```

Which is the same as defining the following mapping using SearchableMetaData explicitly:

```
@Searchable
public class Author {
  @SearchableId
  @SearchableMetaData(name = "id")
  private Long id;
  // ...
}
```

And here is the same mappings as above using xml:

```
<class name="Author" alias="author">
  <id name="id">
    <meta-data>id</meta-data>
  </id>
  <!-- ... -->
</class>
```

6.3.2. Searchable Id Component

A searchable id component represent a composite object acting as the id of a object. It works in a similar manner to searchable component except that it will act as the id of the class.

Here is an example of defining a Searchable Id Component using annotations (note, in this case, B is not a root searchable class, and it needs to define only ids):

```
@Searchable
public class A {
  @SearchableIdComponent
  private B b;
  // ...
}

@Searchable(root = false)
public class B {
  @SearchableId
  private long id1;

  @SearchableId
  private long id2;
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id-component name="b" />
  <!-- ... -->
</class>
<class name="B" alias="b" root="false">
  <id name="id1" />
  <id name="id2" />
</class>
```

6.3.3. Searchable Parent

Searchable Parent mapping provides support for cyclic mappings for components (though bi directional component mappings are also supported). 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.

Here is an example of defining a Searchable Component and Searchable Parent using annotations (note, in this case, B is not a root searchable class, and need not define any ids):

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableComponent
  private B b;
  // ...
}

@Searchable(root = false)
public class B {
  @SearchableParent
  private A a;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id name="id" />
  <component name="b" />
  <!-- ... -->
</class>
<class name="B" alias="b" root="false">
  <parent name="a" />
  <!-- ... -->
</class>
```

6.3.4. Searchable Property and Searchable Meta Data

A Searchable Property maps to a Class attribute/property which is a simple relationship. The searchable property maps to a class attribute that ends up as a String within the search engine. This include primitive types, primitive wrapper types, java.util.Date, java.util.Calendar and many more types that are automatically supported by Compass (please see the converter section). A user defined type can be used as well using a custom converter (though most times, a component relationship is more suited - explained later). A Searchable Mata Data uses the Searchable Property value (converted String value using its registered converter) and stores it in the index against a name.

When using xml mapping, one or more meta-data elements can be defined for a property mapping. When using annotation, a SearchableProperty needs to be defined on the mapped class attribute. A SearchableMetaData annotation can be explicitly defined, as well as SearchableMetaDatas (for multiple meta data). A

SearchableProperty will automatically create a SearchableMetaData (in order not to get annotation hell) if no SearchableMetaData(s) annotation is defined, or a its name is explicitly defined (note, all the SearchableMetaData options are also defined on the SearchableProperty, they apply to the automatically created SearchableMetaData).

Here is an example of defining a Searchable Property using annotations. This example will automatically create a Searchable Meta Data with the name of value (the class field name).

```
@Searchable
public class Author {
  // ...
  @SearchableProperty
  private String value;
  // ...
}
```

This mapping is the same as defining the following annotation using SearchableMetaData explicitly:

```
@Searchable
public class Author {
  // ...
  @SearchableProperty
  @SearchableMetaData(name = "value")
  private String value;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="Author" alias="author">
  <! ... -->
  <property name="value">
    <meta-data>value</meta-data>
  </property>
  <!-- ... -->
</class>
```

6.3.5. Searchable Constant

Searchable Constant allows to define constant meta data associated with a searchable class with a list of values set against a constant name. This is useful for adding static meta-data against a Searchable Class, allowing to create semantic groups across the searchable classes.

Here is how a searchable constant meta-data can be defined using annotations:

```

@Searchable
@SearchableConstant(name = "type", values = {"person", "author"})
public class Author {
}
```

And here is how it can be defined using xml mappings:

```
<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>
  <!-- ... -->
</class>
```

```
```

6.3.6. Searchable Dynamic Meta Data

The dynamic meta data mapping allows to define meta-data saved into the search engine as a result of evaluating an expression. The mapping does not map to any class property and acts as a syntactic meta-data (similar to the constant mapping). The value of the dynamic meta-data tag is the expression evaluated by a Dynamic Converter. Compass comes with several built in dynamic converters: el (Jakarta commons el), jexl (Jakarta commons jexl), velocity, ognl, and groovy. When defining the expression, the root class is registered under the `data` key (for libraries that require it).

Here is an example of how to define a searchable dynamic meta-data (with jakarta commons jexl) using annotations (assuming class A has value1 and value2 as class fields):

```
@Searchable
@SearchableDynamicMetaData(name = "test", expression = "data.value + data.value2", converter = "jexl")
public class A {
}
```

And here is the same mapping using xml:

```
<class name="Author" alias="author">
  <id name="id" />
  <dynamic-meta-data name="test" converter="jexl">
      data.value + data.value2
  </dynamic-meta-data>
  <!-- ... -->
</class>
```

6.3.7. Searchable Reference

A searchable reference mapping maps between one root searchable class and the other. The mapping is only used for keeping the relationship "alive" when performing un-marshalling. The marshalling process marshals only the referenced object ids (based on its id mappings) and use it later in the un-marshalling process to load the referenced object from the index.

Cascading is supported when using reference mappings. Cascading can be configured to cascade any combination of create/save/delete operations, or all of them. By default, no cascading will be performed on the referenced object.

In order to identify the referenced class mapping, Compass needs access to its class mapping definition. In most cases there is no need to define the referenced alias that define the class mapping, as Compass can automatically detect it. If it is required, it can be explicitly set on the reference mappings (an example when Compass needs this mapping is when using Collection without generics or when a class has more than one class mapping).

Currently, Compass does not support lazy loading, this means that when loading a searchable class, all its referenced mappings will be loaded as well.

Here is an example of defining a Searchable Reference using annotations:

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableReference
```

```
  private B b;
  // ...
}

@Searchable
public class B {
  @SearchableId
  private Long id;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id name="id" />
  <reference name="b" />
  <!-- ... -->
</class>
<class name="B" alias="b">
  <id name="id" />
  <!-- ... -->
</class>
```

6.3.8. Searchable Component

A searchable component mapping embeds a searchable class within its owning searchable class. The mapping is used to allow for searches that "hit" the component referenced searchable class to return the owning searchable class (or its parent if it also acts a component mapping up until the root object that was saved).

The component referenced searchable class can be either root or not. An example for a non root component can be a Person class (which is root) with a component mapping to a non root searchable class Name (with firstName and lastName fields). An example for a root component can be a Customer root searchable class and an Account searchable class, where when searching for account details, both Account and Customer should return as hits.

Cascading is supported when using component mappings. Cascading can be configured to cascade any combination of create/save/delete operations, or all of them. By default, no cascading will be performed on the referenced object. Cascading can be performed on non root objects as well, which means that a non root object can be "created/saved/deleted" in Compass (using save operation) and Compass will only cascade the operation on its referenced objects without actually performing the operation on the non root object.

In order to identify the referenced component class mapping, Compass needs access to its class mapping definition. In most cases there is no need to define the referenced alias that define the class mapping, as Compass can automatically detect it. If it is required, it can be explicitly set on the reference mappings (an example when Compass needs this mapping is when using Collection without generics or when a class has more than one class mapping).

Here is an example of defining a Searchable Component using annotations (note, in this case, B is not a root searchable class, and need not define any ids):

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableComponent
  private B b;
  // ...
}

@Searchable(root = false)
public class B {
```

```
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id name="id" />
  <component name="b" />
  <!-- ... -->
</class>
<class name="B" alias="b" root="false">
  <!-- ... -->
</class>
```

6.3.9. Searchable Cascade

The searchable cascading mapping allows to define cascading operations on certain properties without explicitly using component/reference/parent mappings (which have cascading option on them). Cascading actually results in a certain operation (save/delete/create) to be cascaded to and performed on the referenced objects.

Here is an example of a Searchable Cascade mapping based on the class language:

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableCascading(cascade = {Cascade.ALL})
  private B b;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id name="id" />
  <cascade name="b" cascade="all" />
  <!-- ... -->
</class>
```

6.3.10. Searchable Analyzer

The searchable analyzer mapping dynamically controls the analyzer that will be used when indexing the class data. 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 searchable analyzer 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.

Here is an example of a Searchable Analyzer mapping based on the class language:

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableAnalyzer
```

```
  private String language;
  // ...
}
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
  <id name="id" />
  <analyzer name="language" />
  <!-- ... -->
</class>
```

6.3.11. Searchable Boost

The searchable boost mapping dynamically controls the boost value associated with the Resource stored. If the mapping is defined, it will override the class mapping boost attribute setting. The value of the property should be convertable to float value.

Here is an example of a Searchable Analyzer mapping based on the class language:

```
@Searchable
    public class A {
      @SearchableId
      private Long id;
      @SearchableBoost(defaultValue = 2.0f)
      private Float value;
      // ...
    }
```

And here is the same mapping definition using xml:

```
<class name="A" alias="a">
      <id name="id" />
      <boost name="value" default="2.0" />
      <!-- ... -->
    </class>
```

6.4. Specifics

6.4.1. Handling Collection Types

Collection (java.util.Collection) based types cab be mapped using Searchable Property, Searchable Component and Searchable Reference. The same mapping declaration should be used, with Compass automatically detecting that a java.util.Collection is being mapped, and applying the mapping definition to the collection element.

When mapping a Collection with a Searchable Property, Compass will try to automatically identify the collection element type if using Java 5 Generics. If Generics are not used, the class attribute should be set with the FQN of the element class. With Searchable Component or Reference Compass will try to automatically identify the referenced mapping if Generics are used. If generics are not used the ref-alias should be explicitly set.

6.4.2. Managed Id

When marshaling an Object into a search engine, Compass might add internal meta-data for certain Searchable Properties in order to properly un-marshall it correctly. Here is an example mapping where an internal meta-data id will be created for the firstName and lastName searchable properties:

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableProperty(name = "name")
  private String lastName;
  @SearchableProperty(name = "name")
  private String firstName;
  @SearchableProperty
  private String birthdate;
}
```

In the above mapping we map firstName and lastName into "name". Compass will automatically create internal meta-data for both firstName and lastName, since if it did not create one, it won't be able to identify which name belongs to which. Compass comes with three strategies for creating internal meta-data:

- *AUTO*: Compass will automatically identify if a searchable property requires an internal meta-data, and create one for it.
- *TRUE*: Compass will always create an internal meta-data id for the searchable property.
- *FALSE*: Compass will not create an internal meta-data id, and will use the first searchable meta-data as the searchable property meta-data identifier.
- *NO*: Compass will not create an internal meta-data id, and will not try to un-marshall this property at all.
- *NO_STORE*: Compass will not create an internal meta-data id if all of its meta-data mappings have store="no". Otherwise, it will be treated as AUTO.

Setting the managed id can be done on several levels. It can be set on the property mapping level explicitly. It can be set on the class level mapping which will then be applied to all the properties that are not set explicitly. And it can also be set globally be setting the following setting `compass.osem.managedId` which will apply to all the classes and properties that do not set it explicitly. By default, it is set to `NO_STORE`.

6.4.3. Handling Inheritance

There are different strategies when mapping an inheritance tree with Compass. The first apply when the inheritance tree is known in advance. If we take a simple inheritance of class A and class B that extends it, here is the annotation mapping that can be used for it:

```
@Searchable
public class A {
  @SearchableId
  private Long id;
  @SearchableProperty
  private String aValue;
}

@Searchable
public class B extends A {
  @SearchableProperty
  private String bValue;
}
```

Compass will automatically identify that B extends A, and will include all of A mapping definitions (note that

Searchable attributes will not be inherited). When using annotations, Compass will automatically interrogate interfaces as well for possible Searchable annotations, as well have the possibility to explicitly define which mappings to extend using the extend attribute (the mappings to extends need not be annotation driven mappings).

When using xml mapping definition, the above inheritance tree can be mapped as follows:

```
<class name="A" alias="a">
  <id name="id" />
  <property name="aValue">
    <meta-data>aValue</meta-data>
  </property>
</class>
<class name="B" alias="b" extends="a">
  <property name="bValue">
    <meta-data>aValue</meta-data>
  </property>
</class>
```

When using extends explicitly (as needed when using xml mappings), a list of the aliases to extend (comma separated) can be provided. All the extended mapping definitions will be inherited except for class mapping attributes.

If the inheritance tree is not known in advance, a poly flag should be set on all the known mapped inheritance tree. Compass will be able to persist unknown classes that are part of the mapped inheritance tree, using the closest searchable mapping definition. Here is an example of three classes: A and B are searchable classes, with B extending A. C extends B but is not a searchable class and we would still like to persist it in the search engine. The following is the annotation mappings for such a relationship:

```
@Searchable(poly = true)
public class A {
  // ...
}

@Searchable(poly = true)
public class B extends A {
  // ...
}

// Note, No Searchable annotation for C
public class C extends B {
  // ...
}
```

And here is the xml mapping definition:

```
<class name="A" alias="a">
  <id name="id" />
  <property name="aValue">
    <meta-data>aValue</meta-data>
  </property>
</class>
<class name="B" alias="b" extends="a">
  <property name="bValue">
    <meta-data>aValue</meta-data>
  </property>
</class>
```

When saving an Object of class C, B mapping definitions will be used to map it to the search engine. When loading it, an instance of class C will be returned, with all of its B level attributes initialized.

6.4.4. Polymorphic Relationships

Polymorphic relationship are applicable when using component or reference mappings. If we take the following polymorphic relationship of a Father class to a Child class, with a Son and Daughter sub classes, the component/reference mapping relationship between Father and Child is actually a relationship between Father and Child, Son and Daughter. The following is how to map it using annotations:

```
@Searchable
public class Father {
  // ...
  @SearchableComponent
  private Child child;
}

@Searchable(poly = true)
public class Child {
  // ...
}

@Searchable(poly = true)
public class Son extends Child {
  // ...
}

@Searchable(poly = true)
public class Daughter extends Child {
  // ...
}
```

Compass will automatically identify that Child mappings has a Son and a Daughter, and will add them to the ref-alias definition of the SearchableComponent (similar to automatically identifying the mapping of Child). Explicit definition of the referenced aliases can be done by providing a comma separated list of aliases (this will disable Compass automatic detection of related classes and will only use the provided list). Note as well, that the Child hierarchy had to be defined as poly.

Here is the same mapping using xml:

```
<class name="Father" alias="father">
  <id name="id" />
  <component name="child" />
</class>
<class name="Child" alias="chlid" poly="true">
  <!-- ... -->
</class>
<class name="Son" alias="son" poly="true" extends="child">
  <!-- ... -->
</class>
<class name="Daughter" alias="daughter" poly="true" extends="child">
  <!-- ... -->
</class>
```

6.4.5. Cyclic Relationships

Compass OSEM fully supports cyclic relationships both for reference and component mappings. Reference mappings are simple, they are simply defined, and Compass would handle everything if they happen to perform a cyclic relationship.

Bi directional component mappings are simple as well with Compass automatically identifying cyclic relationship. A tree based cyclic relationship is a bit more complex (think of a file system tree like relationship). In such a case, the depth Compass will traverse with the component mapping is controlled using the max-depth attribute (defaults to 1).

6.4.6. Annotations and Xml Combined

Compass allows for Annotations and Xml mappings definitions to be used together. Annotations mappings can extend/override usual cpm.xml mapping definition (event extending xml contract mapping). When using annotations, a .cpm.ann.xml can be defined that will override annotations definitions using xml definitions.

6.4.7. Support Unmarshall

Compass adds an overhead both in terms of memory consumption, processing speed and index size (managed ids) when it works in a mode that needs to support un-marshalling (i.e. getting objects back from the search engine). Compass can be configured not to support un-marshalling. In such a mode it will not add any internal Compass information to the index, and will use less memory. This setting can be a global setting (set within Compass configuration), or per searchable class definitions.

Though initially this mode may sounds unusable, it is important to remember that when working with support unmarshall set to false, the application can still use Compass Resource level access to the search engine. An application that works against the database using an ORM tool for example, might only need Compass to index its domain model into the search engine, and display search results. Displaying search results can be done using Resources (many times this is done even when using support for unmarshalling). Create/Delete/Update operations will be done based on ORM based fetched objects, and mirrored (either explicitly or implicitly) to the search engine.

6.4.8. 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.

6.5. Searchable Annotations Reference

All the annotations are documented in Compass javadoc. Please review it for a complete reference of all of Compass Searchable annotations.

6.6. Searchable Xml Reference

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.

6.6.1. compass-core-mapping

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

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

Table 6.1. OSEM Xml Mapping - compass-core-mapping

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

6.6.2. 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"
        support-unmarshall="true|false"
        boost="boost value for the class"
        converter="converter lookup name"
>
    all?,
    sub-index-hash?.
    (id)*,
    parent?,
    (analyzer?),
    (boost?),
    (property|dynamic-meta-data|component|reference|constant)*
</class>
```

Table 6.2. OSEM Xml Mapping - class

Attribute	Description
name	The fully qualified class name (or relative if the package is declared in `compass-core-mapping`).
alias	The alias of the `Resource` 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 `TOKENIZED` properties. Defaults to the `default` analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the `analyzer` mapping (a child mapping of class mapping) (for a property value that controls the analyzer), the analyzer attribute will have no effects.
root (optional, defaults to `true`)	Specifies if the class is a "root" class or not. You should define the searchable class with false if it only acts as mapping definitions for a component mapping.
poly (optional, defaults to `false`)	Specifies if the class will be enabled to support polymorphism. This is the less preferable way to map an inheritance tree, since the extends attribute can be used to statically extend base classes or contracts.
poly-class (optional)	If poly is set to `true`, 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 poly-class is set, the class name will not be saved to the index, and the value of poly-class 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 `class` mapping or a `contract` mapping. Note that can extend more than one `class/contract`
support-unmarhsall (optional)	Controls if the searchable class will support unmarshalling from the search engine or using `Resource` is enough. Un-marshalling is the process of converting a raw `Resource` into the actual domain object. If support un-marshall is enabled extra information will be stored within the search engine, as well as consumes extra memory. Defaults to Compass global setting `compass.osem.supportUnmarshall` (which in turn defaults to `true`).
boost (optional, defaults to `1.0`)	Specifies the boost level for the class.
converter (optional)	The global converter lookup name registered with the configuration. Responsible for converting the `ClassMapping` definition. Defaults to compass internal `ClassMappingConverter`.

Root classes have their own index within the search engine index directory (by default). 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 or the `sub-index-hash` element, otherwise it will create a sub-index based on the alias name.

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.

6.6.3. contract

Declaring a searchable contract using the `contract` element.

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

Table 6.3. OSEM Xml Mapping - contract

Attribute	Description
alias	The alias of the contract. Will be used as the alias name in the `class` 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 than extend the contract within the mapped classes (even if you don't have a concrete interface or class in your Java definition).

6.6.4. 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"
      managed-id-converter="managed id converter lookup name"
      exclude-from-all="no|yes|no_analyzed"
      converter="converter lookup name"
  >
 (meta-data)*
</id>
```

Table 6.4. OSEM Xml Mapping - id

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. `property` access using the Java Bean accessor methods, while `field` directly access the class fields.
boost (optional, default to `1.0f`)	The boost level that will be propagated to all the meta-data defined within the id.
class (optional)	An explicit definition of the class of the property, helps for certain converters.
managed-id (optional, defaults to `auto`)	The strategy for creating or using a class property meta-data id (which maps to a `ResourceProperty`).
managed-id-converter (optional)	The global converter lookup name applied to the generated managed id (if generated).
exclude-from-all (optional, defaults to `no`)	Excludes the class property from participating in the "all" meta-data, unless specified in the meta-data level. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).

Attribute	Description
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`), `String` type, and many other custom types, with the only requirement that a type used for an id will be converted to a single String.

6.6.5. 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"
      override="true|false"
      managed-id="auto|true|false"
      managed-id-index="[compass.managedId.index setting]|no|un_tokenized"
      managed-id-converter="managed id converter lookup name"
      exclude-from-all="no|yes|no_analyzed"
      converter="converter lookup name"
>
   (meta-data)*
</property>
```

Table 6.5. OSEM Xml Mapping - property

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. `property` means accessing using the Java Bean accessor methods, while `field` 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 `java.util.Collection` type properties, since it applies to the collection elements).
analyzer (optional, defaults to the class mapping analyzer decision scheme)	The name of the analyzer that will be used to analyze `TOKENIZED` meta-data mappings defined for the given property. Defaults to the class mapping analyzer decision scheme based on the analyzer set, or the `analyzer` mapping property.
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.
managed-id (optional, defaults to `auto`)	The strategy for creating or using a class property meta-data id (which maps to a `ResourceProperty`.

Attribute	Description
managed-id-index (optional, defaults to `compass.managedId.index` setting, which defaults to `no`)	Can be either `un_tokenized` or `no`. 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 `un_tokenized`).
managed-id-converter (optional)	The global converter lookup name applied to the generated managed id (if generated).
exclude-from-all (optional, defaults to `no`)	Excludes the class property from participating in the "all" meta-data, unless specified in the meta-data level. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).
converter (optional)	The global converter lookup name registered with the configuration.

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 (unless you are using generics).

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`).

6.6.6. 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 6.6. OSEM Xml Mapping - analyzer

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. `property` means accessing using the Java Bean accessor methods, while `field` directly accesses the class fields.
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 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.

6.6.7. boost

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

```
<boost
      name="property name"
      default="the boost default value when no property value is present"
      accessor="property|field"
      converter="converter lookup name"
>
</boost>
```

Table 6.7. OSEM Xml Mapping - analyzer

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. `property` means accessing using the Java Bean accessor methods, while `field` directly accesses the class fields.
default (optional, defaults `1.0f`)	The default value if the property has a null value.
converter (optional)	The global converter lookup name registered with the configuration.

The boost class property mapping, controls the boost associated with the Resource created based on the mapped property. The value of the property should be allowed to be converted to float.

6.6.8. 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"
      null-value="String value that will be stored when the property value is null"
      exclude-from-all="[parent's exclude-from-all]|no|yes|no_analyzed"
      converter="converter lookup name"
      term-vector="no|yes|positions|offsets|positions_offsets"
      format="the format string (only applies to formatted elements)"
>
</meta-data>
```

Table 6.8. OSEM Xml Mapping - meta-data

Attribute	Description
store (optional, defaults to `yes`)	If the value of the class property that the meta-data maps to, is going to be stored in the index.
index (optional, defaults to `tokenized`)	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 (`tokenized`), or is going to be used as is (`un_tokenized`).
boost (optional, defaults to `1.0f`)	Controls the boost level for the `meta-data`.
analyzer (optional, defaults to the parent analyzer)	The name of the analyzer that will be used to analyze `TOKENIZED` 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 `analyzer` mapping property.
term-vector (optional, defaults to `no`)	The term vector value of meta data.
reverse (optional, defaults to `no`)	The meta-data will have it's value reversed. Can have the values of `no` - no reverse will happen, `string` - the reverse will happen and the value stored will be a reversed string, and `reader` - a special reader will wrap the string and reverse it. The `reader` option is more performant, but the `store` and `index` settings will be discarded.
exclude-from-all (optional, defaults to the parent's `exclude-from-all` value)	Excludes the meta-data from participating in the "all" meta-data. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).
null-value (optional, defaults to not storing the anything on null)	A String null value that will be used when the property evaluates to null.
converter (optional)	The global converter lookup name registered with the configuration. Note, that in case of a `Collection` property, the converter will be applied to the collection elements (Compass has it's own converter for Collections).
format (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.

6.6.9. dynamic-meta-data

Declaring and using the `dynamic-meta-data` element.

```
<dynamic-meta-data
      name="The name the meta data will be saved under"
      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"
      null-value="optional String value when expression is null"
      exclude-from-all="[parent's exclude-from-all]|no|yes|no_analyzed"
      converter="the Dynamic Converter lookup name (required)"
      format="the format string (only applies to formatted elements)"
>
</meta-data>
```

Table 6.9. OSEM Xml Mapping - dynamic-meta-data

Attribute	Description
name	The name the dynamic meta data will be saved under (similar to the tag name of the `meta-data` mapping).
store (optional, defaults to `yes`)	If the value of the class property that the meta-data maps to, is going to be stored in the index.
index (optional, defaults to `tokenized`)	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 (`tokenized`), or is going to be used as is (`un_tokenized`).
boost (optional, defaults to `1.0f`)	Controls the boost level for the `meta-data`.
analyzer (optional, defaults to the parent analyzer)	The name of the analyzer that will be used to analyze `TOKENIZED` 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 `analyzer` mapping property.
reverse (optional, defaults to `no`)	The meta-data will have it's value reversed. Can have the values of `no` - no reverse will happen, `string` - the reverse will happen and the value stored will be a reversed string, and `reader` - a special reader will wrap the string and reverse it. The `reader` option is more performant, but the `store` and `index` settings will be discarded.
exclude-from-all (optional, defaults to the parent's `exclude-from-all` value)	Excludes the meta-data from participating in the "all" meta-data. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).
null-value (optional, defaults to not saving the value)	If the expression evaluates to null, the String null value that will be stored for it.
converter (required)	The global dynamic converter lookup name registered with the configuration. Built in dynamic converters include: el, jexl, velocity, ognl and groovy.
format (optional)	Allows for quickly setting a format for format-able types (dates, and numbers), without creating/registering a specialized converter under a lookup name. Applies when the dynamic expression

Attribute	Description
	evaluates to a formatable object. Must set the type attribute as well.
type (optional)	The fully qualified class name of the object evaluated as a result of the dynamic expression. Applies when using formats.

The dynamic meta data mapping allows to define meta-data saved into the search engine as a result of evaluating an expression. The mapping does not map to any class property and acts as a syntactic meta-data (similar to the constant mapping). The value of the dynamic meta-data tag is the expression evaluated by a Dynamic Converter. Compass comes with several built in dynamic converters: el (Jakarta commons el), jexl (Jakarta commons jexl), velocity, ognl, and groovy. When defining the expression, the root class is registered under the `data` key (for libraries that require it).

6.6.10. 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"
      cascade="comma separated list of create,save,delete or all"
>
</component>
```

Table 6.10. OSEM Xml Mapping - component

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 component. This is an optional attribute since under most conditions, Compass can infer the referenced alias (it actually can't infer it when using Collection without generics, or when a class has more than one mapping). In case of polymorphic relationship, a list of aliases can be provided (though again, Compass will try and auto detect the list of aliases if none is defined).
max-depth (optional, defaults to 1)	The depth of cyclic component mappings allowed.
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.
accessor (optional, defaults to `property`)	The strategy to access the class property value. `property` access using the Java Bean accessor methods, while `field` directly access the class fields.
converter (optional)	The global converter lookup name registered with the configuration.
cascade (optional, defaults to none)	A comma separated list of operations to cascade. The operations

Attribute	Description
	names are: create, save and delete. all can be used as well to mark cascading for all operations.

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`.

6.6.11. 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"
        cascade="comma separated list of create,save,delete or all"
  >
</reference>
```

Table 6.11. OSEM Xml Mapping - reference

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 most conditions, Compass can infer the referenced alias (it actually can't infer it when using Collection without generics, or when a class has more than one mapping). In case of polymorphic relationship, a list of aliases can be provided (though again, Compass will try and auto detect the list of aliases if none is defined).
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 root="false") 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 ref-comp-alias you will have to save all the relevant encompassing classes.
accessor (optional, defaults to `property`)	The strategy to access the class property value. `property` access using the Java Bean accessor methods, while `field` directly access the class fields.

Attribute	Description
converter (optional)	The global converter lookup name registered with the configuration.
cascade (optional, defaults to none)	A comma separated list of operations to cascade. The operations names are: create, save and delete. all can be used as well to mark cascading for all operations.

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

6.6.12. parent

Declaring and using the `parent` element.

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

Table 6.12. OSEM Xml Mapping - parent

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. `property` access using the Java Bean accessor methods, while `field` 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 (though bi directional component mappings are also supported). 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.

6.6.13. constant

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

```
<constant
          exclude-from-all="no|yes|no_analyzed"
          converter="converter lookup name"
    >
   meta-data,
   meta-data-value+
</reference>
```

Table 6.13. OSEM Xml Mapping - constant

Attribute	Description
exclude-from-all (optional, defaults to `false`)	Excludes the constant meta-data and all it's values from participating in the "all" feature. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).
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 7. XSEM - Xml to Search Engine Mapping

7.1. Introduction

Compass 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.

7.2. 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(
AliasedXmlObject xmlObject = NodeAliasedXmlObject(alias, doc);
session.save(xmlObject);
```

7.3. 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 7.1. Compass XmlContentConverters

XmlContentConverter	Description
org.compass.core.xml.javax.converter.NodeXmlContentConverter	Support for JSE 5 xml libraries. Not recommended on account of performance.
org.compass.core.xml.dom4j.converter.SAXReaderXmlContentConverter	Support dom4j `SAXReader` for parsing, and `XMLWriter` to write the raw xml data.
org.compass.core.xml.dom4j.converter.XPPReaderXmlContentConverter	Support dom4j `XPPReader` for parsing, and `XMLWriter` to write the raw xml data.
org.compass.core.xml.dom4j.converter.XPP3ReaderXmlContentConverter	Support dom4j `XPP3Reader` for parsing, and `XMLWriter` to write the raw xml data.
org.compass.core.xml.dom4j.converter.STAXReaderXmlContentConverter	Support dom4j `STAXEventReader` for parsing, and `XMLWriter` 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="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.

7.4. 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 used 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 onc the the registered converter creates, and not the raw xml object.

7.5. 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="data11attr">data11</data1>
```

```
        <data1 value="data12attr">data12</data1>
    </data>
    <data>
        <id value="2"/>
        <data1 value="data21attr">data21</data1>
        <data1 value="data22attr">data22</data1>
    </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 2.0//EN"
      "http://www.compass-project.org/dtd/compass-core-mapping-2.0.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.

7.5.1. 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?,
     sub-index-hash?,
     xml-id*,
     (xml-analyzer?),
     (xml-boost?),
     (xml-property)*,
     (xml-content?)
```

Table 7.2. xml-object mapping

Attribute	Description
alias	The name of the alias that represents the `XmlObject`.
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 `Resources` saved to the search engine.
analyzer (optional, defaults to the `default` analyzer)	The name of the analyzer that will be used to analyze `TOKENIZED` properties. Defaults to the `default` analyzer which is one of the internal analyzers that comes with Compass. Note, that when using the `xml-analyzer` mapping (a child mapping of xml object mapping) (for an xml element that controls the analyzer), the analyzer attribute will have no effects.

7.5.2. 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 7.3. xml-id mapping

Attribute	Description
name	The name of the `xml-id`. Will be used when constructing the xml-id 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

Attribute	Description
	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.

7.5.3. 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="no|yes|no_analyzed"
      value-converter="value converter lookup name"
      converter="converter lookup name"
/>
```

Table 7.4. 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 `yes`)	If the value of the xml property is going to be stored in the index.
index (optional, defaults to `tokenized`)	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 (`tokenized`), or is going to be used as is (`un_tokenized`).
boost (optional, defaults to `1.0f`)	Controls the boost level for the xml property.
analyzer (optional, defaults to the xml mapping analyzer decision scheme)	The name of the analyzer that will be used to analyze `TOKENIZED` xml property mappings defined for the given property. Defaults to the xml mapping analyzer decision scheme based on the analyzer set, or the `xml-analyzer` mapping.
exclude-from-all (optional, default to	Excludes the property from participating in the "all" meta-data. If

Attribute	Description
no)	set to no_analyzed, un_tokenized properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).
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 no - no reverse will happen, string - the reverse will happen and the value stored will be a reversed string, and reader - a special reader will wrap the string and reverse it. The reader option is more performant, but the store and index 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). 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-property mapping.

7.5.4. 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 7.5. xml-analyzer mapping

Attribute	Description
name	The name of the xml-analyzer (results in a Property).
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 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.

7.5.5. xml-boost

Declaring a dynamic boost mapping controlling the boost level using the `xml-boost` element.

```
<xml-analyzer
     name="property name"
     xpath="xpath expression"
     default="the boost default value when no property value is present"
     converter="converter lookup name"
>
</xml-analyzer>
```

Table 7.6. xml-analyzer mapping

Attribute	Description
name	The name of the xml-analyzer (results in a `Property`).
xpath	The xpath expression used to identify the xml-analyzer. Must return a single xml element.
default (optional, defaults to `1.0`)	The default boost value if no value is found.
converter (optional)	The global converter lookup name registered with the configuration.

The boost xml property mapping, controls the boost associated with the Resource created based on the mapped property. The value of the property should be allowed to be converted to float.

7.5.6. 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 7.7. xml-content mapping

Attribute	Description
name	The name the xml content will be saved under.

Attribute	Description
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 unmarshall 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 8. RSEM - Resource/Search Engine Mapping

8.1. Introduction

Compass provides OSEM technology for use with an applications Object domain model or XSEM when working with xml data structures. Compass also provides Resource Mapping technology for resources other than Objects/XML (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 additional features (i.e. transactions, fast updates). Working with `Resources` makes your migration easy (as it is similar to working with Lucene Document).
- 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.

8.2. 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 (this is needs especially for update/delete operations).

Here is an example of a Resource Mapping XML document:

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
    "-//Compass/Compass Core Mapping DTD 2.0//EN"
    "http://www.compass-project.org/dtd/compass-core-mapping-2.0.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" converter="my-date" />
  </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.

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

session.save(r);
```

The Resource Mapping file example above defines mappings for three resource types (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.

Here is an example of how `resource-property` mappings can simplify Resource construction code:

```
ResourceFactory resourceFactory = compass.getResourceFactory();
Resource r = resourceFactory.createResource("c");
r.addProperty("id", 1);
r.addProperty("value1", "this is a sample value");
r.addProperty("value5", new Date()); // will use the my-date converter (using the format defined there)
session.save(r);
```

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.

8.2.1. 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?,
    sub-index-hash?,
    resource-id*,
    (resource-analyzer?),
    (resource-boost?),
    (resource-property)*
```

Table 8.1.

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

8.2.2. 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 8.2.

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

8.2.3. 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 8.3.

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

8.2.4. 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="no|yes|no_analyzed"
      converter="converter lookup name"
>
</resource-property>
```

Table 8.4.

Attribute	Description
name	The name of the `Property` (known also as the name of the meta-data).
store (optional, defaults to `yes`)	If the value of the resource property is going to be stored in the index.
index (optional, defaults to `tokenized`)	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 (`tokenized`), or is going to be used as is (`un_tokenized`).
boost (optional, defaults to `1.0f`)	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 `TOKENIZED` resource property mappings defined for the given property. Defaults to the resource mapping analyzer decision scheme based on the analyzer set, or the `resource-analyzer` mapping.
exclude-from-all (optional, default to `false`)	Excludes the property from participating in the "all" meta-data. If set to `no_analyzed`, `un_tokenized` properties will be analyzed when added to the all property (the analyzer can be controlled using the analyzer attribute).

Attribute	Description
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 `no` - no reverse will happen, `string` - the reverse will happen and the value stored will be a reversed string, and `reader` - a special reader will wrap the string and reverse it. The `reader` option is more performant, but the `store` and `index` 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 `resourcde-property` mapping.

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

8.2.5. 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 8.5.

Attribute	Description
name	The name of the `Property` (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.

8.2.6. resource-boost

Declaring a dynamic property to control the resource boost value using the `resource-boost` element.

```
<resource-boost
      name="property name"
      default="the boost default value when no property value is present"
      converter="converter lookup name"
>
</resource-boost>
```

Table 8.6.

Attribute	Description
name	The name of the `Property` (known also as the name of the meta-data).
default (optional, defaults to `1.0`)	The default value if the property has a null value.
converter (optional)	The global converter lookup name registered with the configuration.

The boost resource property mapping, controls the boost associated with the Resource created based on the mapped property. The value of the property should be allowed to be converted to float.

Chapter 9. Common Meta Data

9.1. 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.

9.2. Commnon 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 2.0//EN"
    "http://www.compass-project.org/dtd/compass-core-meta-data-2.0.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 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>
```

9.3. 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 2.0//EN"
    "http://www.compass-project.org/dtd/compass-core-mapping-2.0.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>
```

9.4. Commnon 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="classpathhref"/>
<mdtask destdir="${java.src.dir}">
    <fileset dir="${java.src.dir}">
        <include name="**/*"/>
    </fileset>
</mdtask>
```

Chapter 10. Transaction

10.1. Introduction

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

10.2. Session Lifecycle

`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, Spring, Hibernate 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.

When using the `openSession` method, Compass will automatically try and join an already running outer transaction. An outer transaction can be an already running local Compass transaction, a JTA transaction, a Hibernate transaction, or a Spring managed transaction. If Compass manages to join an existing outer transaction, the application does not need to call `CompassSession#beginTransaction()` or use `CompassTransaction` to manage the transaction (since it is already managed). This allows to simplify the usage of Compass within managed environments (CMT or Spring) where a transaction is already in progress by not requiring explicit Compass code to manage a Compass transaction. In fact, calling `beginTransaction` will not actually begin a transaction in such a case, but will simply join it (with only the rollback method used).

10.3. 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`.

10.4. JTA Synchronization Transaction

Compass 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. It can be configured to perform the commit in the `beforeCompletion` phase, which is useful when storing the index in the database.

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 can also set the transaction manager lookup based on the environment your application will be running at (Compass will try to automatically identify it).

10.5. XA Transaction

Compass provides support for JTA transactions by enlisting an `XAResource` with a currently active `Transaction`. This allows for Compass to participate in a two phase commit process. 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).

The XA support provided allows for proper two phase commit transaction operations, but do not provide a full implementation such as a JCA implementation (mostly for recovery).

In order to configure Compass to work with the JTA XA Transaction, you must set the `compass.transaction.factory` to `org.compass.core.transaction.XATransactionFactory`. You can also set the transaction manager lookup based on the environment your application will be running at (Compass will try to automatically identify it).

Chapter 11. Working with objects

11.1. Introduction

Lets assume you have download and configured Compass within your application and create some RSEM/OSEM/XSEM 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 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 internal representation (`Resource`) or application domain Objects.

11.2. 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);
```

When using OSEM and defining cascading on component/reference mappings, Compass will cascade save operations to the target referenced objects (if they are marked with save cascade). Non root objects are allowed to be saved in Compass if they have cascading save relationship defined.

11.3. 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.

11.4. 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).

```
session.delete(Author.class, 12);
// or :
session.delete(Author.class, new Author(12));
// or :
session.delete(Author.class, "12"); // Everything in the search engine is a String at the end
```

When using OSEM and defining cascading on component/reference mappings, Compass will cascade delete operations to the target referenced objects (if they are marked with delete cascade). Non root objects are allowed to be deleted in Compass if they have cascading save relationship defined. Note, deleting objects by their id will not cause cascaded relationships to be deleted, only when the actual object is passed to be deleted, with the relationships initialized (the object can be loaded from the search engine).

11.5. 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");
```

11.5.1. 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 11.1.

Expression	Hits That
jack	Contain the term `jack` in the default search field
jack london (jack AND london)	Contains the term `jack` and `london` in the default search field
jack OR london	Contains the term `jack` or `london`, or both, in the default search field
+jack +london (jack AND london)	Contains both `jack` and `london` in the default search field
name:jack	Contains the term `jack` in the `name` property (meta-data)
name:jack -city:london (name:jack AND NOT city:london)	Have `jack` in the `name` property and don't have `london` in the `city` property
name:"jack london"	Contains the exact phrase `jack london` in the `name` property
name:"jack london"~5	Contain the term `jack` and `london` within five positions of one another
jack*	Contain terms that begin with `jack`
jack~	Contains terms that are close to the word `jack`
birthday:[1870/01/01 TO 1920/01/01]	Have the `birthday` 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.

11.5.2. Query String - Range Queries Extensions

Compass simplifies the usage of range queries when working with dates and numbers. When using numbers it is preferred to store the number if a lexicography correct value (such as 00001, usually using the format attribute). When using range queries, Compass allows to execute the following query: value:[1 TO 3] and internally Compass will automatically translate it to value:[0001 TO 0003].

When using dates, Compass allows to use several different formats for the same property. The format of the Date object should be sortable in order to perform range queries. This means, for example, that the format attribute should be: format="yyyy-MM-dd". This allows for range queries such as: date:[1980-01-01 TO 1985-01-01] to work. Compass also allows to use different formats for range queries. It can be configured within the format configuration: format="yyyy-MM-dd||dd-MM-yyyy" (the first format is the one used to store the String). And now the following range query can be executed: date:[01-01-1980 TO 01-01-1985].

Compass also allows for math like date formats using the now keyword. For example: "now+1year" will translate to a date with a year from now. For more information please refer to the DateMathParser javadoc.

11.5.3. 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 11.2.

Method	Description
`getLength()` or `length()`	Number of resources in the hits.
`score(n)`	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.
`resource(n)`	Resource instance of the n'th top-scoring resource.
`data(n)`	Object instance of the n'th top-scoring resource.

11.5.4. 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 than 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();

// Remmember, 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();
```

When using query strings and query parsers, Compass enhances Lucene query parser to support custom formats (for dates and numbers, for example) as well as support dot path notation. The query: `a.familyname.family-name:london` will result in a query matching on `familyName` to `london` as well as wrapping the query with one that will only match the `a` alias.

11.5.5. Terms and Frequencies

Compass allows to easily get all the terms (possible values) for a property / meta-data name and their respective frequencies. This can be used to build a frequency based list of terms showing how popular are different tags (as different blogging sites do for example). Here is a simple example of how it can be used:

```
CompassTermFreq[] termFreqs = session.termFreqsBuilder(new String[]{"tag"}).toTermFreqs();
// iterate over the term freqs and display them

// a more complex example:
termFreqs = session.termFreqsBuilder(new String[]{"tag"}).setSize(10).
        .setSort(CompassTermFreqsBuilder.Sort.TERM).normalize(0, 1).toTermFreqs();
```

11.5.6. CompassSearchHelper

Compass provides a simple search helper providing support for pagination and automatic hits detach. The search helper can be used mainly to simplify search results display and can be easily integrated with different MVC frameworks. CompassSearchHelper is thread safe. Here is an example of how it can be used:

```
// constructs a new search helper with page size 10.
CompassSearchHelper searchHelper = new CompasssSearchHelper(compass, 10);
// ...
CompassSearchResults results = searchHelper.search(new CompassSearchCommand("test", new Integer(0)));
for (int i = 0; i < results.getHits().length; i++) {
  CompassHit hit = results.getHits()[i];
  // display the results
}
// iterate through the search results pages
for (int i = 0; i < results.getPages().length; i++) {
  Page page = results.getPages()[i];
  // display a page, for example 1-10, 11-20, 21-30
}
```

11.5.7. 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 12. 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 13. 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 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 14. Introduction

14.1. Overview

Compass Gps provides integration with different indexable data sources using two interfaces: *CompassGps* and *CompassGpsDevice*. Both 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.

Compass Gps is built on top of Compass Core module, 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.

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).

14.2. CompassGps

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

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

14.2.1. 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 Single Compass Gps will clone the provided `Compass` instance. Additional or overriding settings can be provided using indexSettings. By default, default overriding settings are: batch_insert as transaction isolation mode, and disabling of any cascading operations (as they usually do not make sense for index operations). A prime example for overriding setting of the index operation can be when using a database as the index storage, but define a file based storage for the index operation (the index will be built on the file system and then copied to the database).

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.

14.2.2. 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.

14.3. 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* 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.

14.3.1. 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.

14.4. 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.

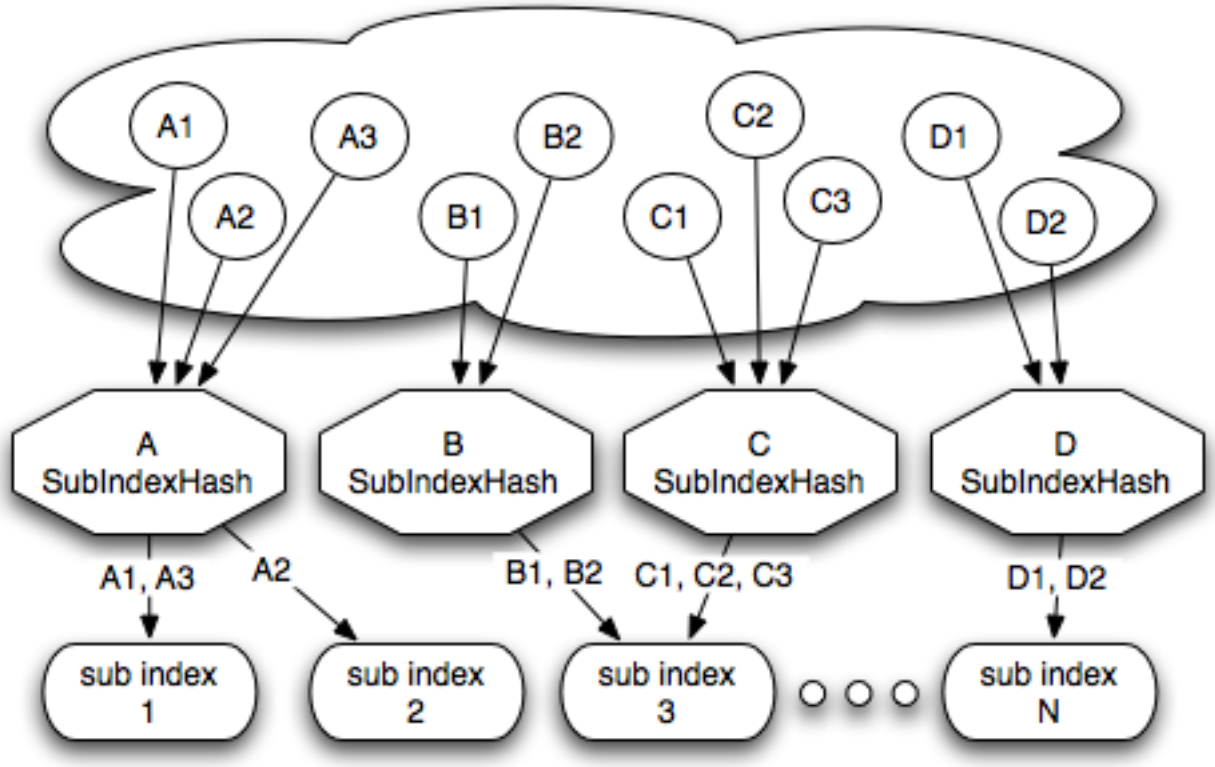
The following code snippet shows how to configure Compass Gps as well as managing it's lifecycle.

```
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();
```

14.5. Parallel Device

The Compass Gps module provides a convenient base class for parallel indexing of devices (data sources). The `AbstractParallelGpsDevice` and its supporting classes allow to simplify paralleled gps devices index operations (and is used by Hibernate and Jpa Gps devices).

If we use the following aliases mapped to different sub indexes as an example:

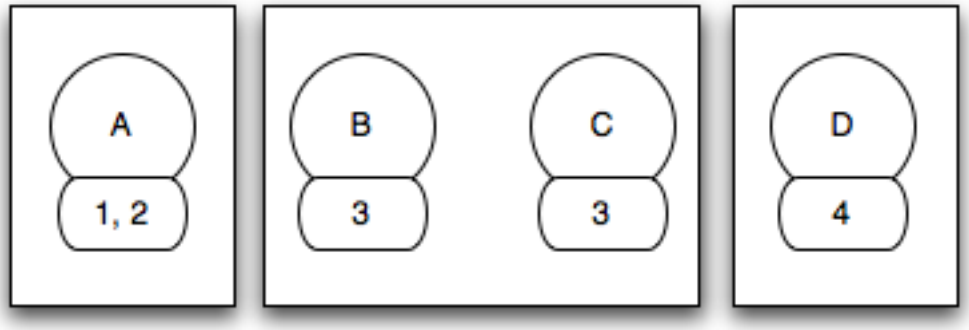


Alias To Sub Index Mapping

The first step during the parallel device startup (`start` operation) is to ask its derived class for its indexable entities (the parallel device support defines an index entity as an entity "template" about to be indexed associated with a name and a set of sub indexes). In our case, the following are the indexed entities:

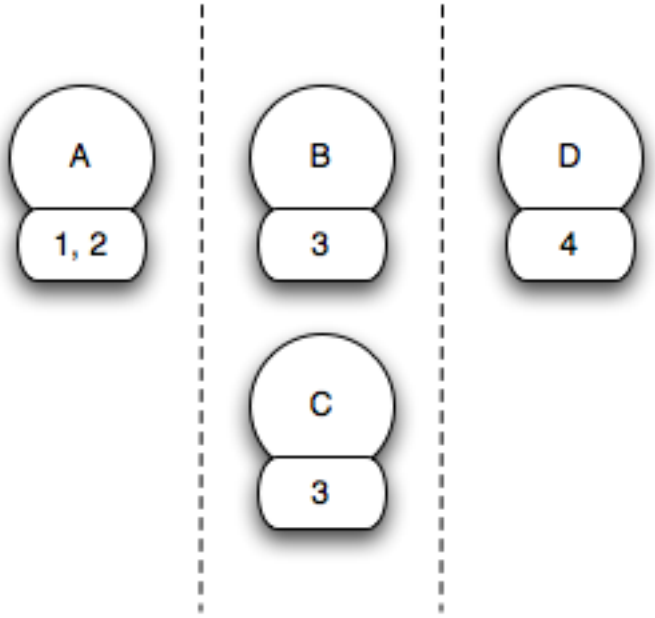
Parallel Index Entities

Then, still during the startup process, the index entities are partitioned using an `IndexEntitiesPartitioner` implementation. The default (and the only one provided built in) is the `SubIndexIndexEntitiesPartitioner` that partitions the entities based on their sub index allocation (this is also usually the best partitioning possible, as locking is performed on the sub index level). Here are the index entities partitioned:



Partitioned Index Entities

During the index operation, a `ParallelIndexExecutor` implementation will then execute the index operation using the partitioned index entities, and an `IndexEntitiesIndexer` implementation (which is provided by the derived class). The default implementation is `ConcurrentParallelIndexExecutor` which creates N threads during the index operation based on the number of partitioned entities and then executes the index process in parallel on the partitioned index entities. In our case, the following diagram shows the index process:



Concurrent Parallel Index Process

Compass also comes with a simple `SameThreadParallelIndexExecutor` which basically uses the same thread of execution to execute the index operation sequentially.

14.6. 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 sorce 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 controll 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 15. JDBC

15.1. 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.

15.2. 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.

15.2.1. 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 it's 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.

15.2.2. 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.

15.3. 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).

15.3.1. 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.ve
  + "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).

15.3.2. 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.

15.3.3. 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"));
```

15.4. 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));
```

15.5. 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 16. Embedded Hibernate

16.1. Introduction

Compass allows for embedded integration with Hibernate and Hibernate JPA. Using simple configuration, Compass will automatically perform mirroring operations (mirroring changes done through Hibernate to the search engine), as well as allow to simply index the content of the database using Hibernate.

The integration involves few simple steps. The first is enabling Embedded Compass within Hibernate. If Hibernate Annotations or Hibernate EntityManager (JPA) are used, just dropping Compass jar file to the classpath will enable it (make sure you don't have Hibernate Search in the classpath, as it uses the same event class name :)). If Hibernate Core is used, the following event listeners need to be configured:

```
<hibernate-configuration>
<session-factory>

  <!-- ... -->

  <event type="post-update">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
  <event type="post-insert">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
  <event type="post-delete">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
  <event type="post-collection-recreate">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
  <event type="post-collection-remove">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
  <event type="post-collection-update">
      <listener class="org.compass.gps.device.hibernate.embedded.CompassEventListener"/>
  </event>
</session-factory>
</hibernate-configuration>
```

Now that Compass is enabled with Hibernate there is one required Compass property in order to configure it which is the location of where the search engine index will be stored. This is configured as a Hibernate property configuration using the key `compass.engine.connection` (for example, having the value `file://tmp/index`). When it is configured, Compass will automatically use the mapped Hibernate classes and check if one of them is searchable. If there is at least one, then the listener will be enabled. That is it!. Now, every operation done using Hibernate will be mirrored to the search engine.

Direct access to the Compass (for example to execute search operations), either the `HibernateHelper` (when using pure Hibernate) or `HibernateJpaHelper` (when using Hibernate JPA) can be used to access it. For example:

```
Compass compass = HibernateHelper.getCompass(sessionFactory);
CompassSession session = compass.openSession();
CompassTransaction tr = session.beginTransaction();

CompassHits hits = session.find("search something")

tr.commit();
session.close();
```

In order to completely reindex the content of the database based on both the Hibernate and Compass mappings,

a Compass Gps can be created. Here is an example of how to do it:

```
CompassGps gps = HibernateHelper.getCompassGps(sessionFactory);
gps.index();
```

16.2. Configuration

The basic configuration of embedded Hibernate is explained in the introduction section. Within the Hibernate (or JPA persistance xml) configuration, the Compass instance used for mirroring and searching can be configured using Compass usual properties (using the `compass.` prefix). If configuring Compass using external configuration is needed, the `compass.hibernate.config` can be used to point to Compass configuration file.

An implementation of `HibernateMirrorFilter` can also be configured in order to allow for filtering out specific objects from the index (for example, based on their specific content). The `compass.hibernate.mirrorFilter` property should be configured having the fully qualified class name of the mirroring filter implementation.

The Compass instance created automatically for the indexing operation can be also configured using specific properties. This properties should have the prefix of `gps.index.`. This is usually configured to have specific parameters for the indexing Compass, for example, having a different index storage location for it while indexing.

16.3. Transaction Management

Compass will integrate with Hibernate transaction management (using whichever transaction management it does) by default. When configuring Compass to work with JTA Sync or XA, Compass will integrate with these transaction management.

Chapter 17. Hibernate

17.1. 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).

Hibernate Gps Device extends Compass Gps `AbstractParallelGpsDevice` and supports parallel index operations. It is discussed in more detail here: Section 14.5, “Parallel Device”.

17.2. Configuration

When configuring the Hibernate device, one must instantiate `HibernateGpsDevice`. After instantiating the device, it must be initialized with a Hibernate `SessionFactory`.

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

```
Compass compass = ... // set compass instance
SingleCompassGps gps = new SingleCompassGps(compass);
CompassGpsDevice hibernateDevice = new HibernateGpsDevice("hibernate", sessionFactory);
gps.addDevice(hibernateDevice);
.... // configure other devices
gps.start();
```

In order to register event listener with Hibernate `SessionFactory`, the actual instance of the session factory need to be obtained. The Hibernate device allows for a pluggable `NativeHibernateExtractor` implementation responsible for extracting the actual instance. Compass comes with a default implementation when working within a Spring environment called: `SpringNativeHibernateExtractor`.

17.2.1. Deprecated Hibernate Devices

For backward compatibility, Compass supports previous `Hibernate2GpsDevice` and `Hibenrate3GpsDevice`. The classes have moved to a different package, and are usable with a simple change to the package name. The new package for the deprecated devices is: `org.compass.gps.device.hibernate.dep`.

17.2.1.1. 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
SingleCompassGps 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();
```

17.3. Index Operation

Hibernate Gps device provides the ability to index a database. 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.

The indexing process is pluggable and Compass comes with two implementations. The first, `PaginationHibernateIndexEntitiesIndexer`, uses `setFirstResult` and `setMaxResults` in order to perform pagination. The second one, `ScrollableHibernateIndexEntitiesIndexer`, uses Hibernate scrollable resultset in order to index the data. The default indexer used is the scrollable indexer.

During the indexing process Compass will execute a default query which will fetch all the relevant data from the database using Hibernate. The query itself can be controlled both by setting a static sql query and providing a query provider. This setting applies per entity. Note, when using the scrollable indexer, it is preferable to use a custom query provider that will return specific Hibernate `Criteria` instead of using static sql query.

17.4. 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

17.5. HibernateSyncTransaction

Compass integrates with Hibernate transaction synchronization services. This means that whichever Hibernate transaction management is used (Jta, JDBC, ...) you are using, the `HibernateSyncTransaction` will synchronize with the transaction upon transaction completion. The Hibernate transaction support uses Hibernate context session in order to obtain the current session and the current transaction. The application

using this feature must also use Hibernate context session (which is the preferred Hibernate usage model starting from Hibernate 3.2).

If you are using the `HibernateSyncTransaction`, a Hibernate based transaction must already be started in order for HibernateSyncTransaction to join. If no transaction is started, Compass can start one (and will commit it eventually). Note, if you are using other transaction management abstraction (such as Spring), it is preferable to use it instead of this transaction factory.

In order to configure Compass to work with the `HiberanteSyncTransaction`, you must set the `compass.transaction.factory` to `org.compass.gps.device.hiberante.transaction.HibernateSyncTransactionFactory`. Additional initialization should be performed by calling `HibernateSyncTransactionFactory.setSessionFactory` with Hibernate `SessionFactory` instance before the `Compass` is created.

17.6. Hibernate Transaction Interceptor

When working with Hibernate transactions (and not utilizing Hibernate context session) and Compass local transactions, an Compass implementation of Hibernate `Interceptor` can be used to synchronize with a Hibernate session. `CompassTransactionInterceptor` can be used to inject an instance of itself into Hibernate `SessionFactory`. Please refer to its javadoc for more information.

Chapter 18. JPA (Java Persistence API)

18.1. 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).

JPA Gps Device extends Compass Gps `AbstractParallelGpsDevice` and supports parallel index operations. It is discussed in more detail here: Section 14.5, “Parallel Device”.

18.2. 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 it's 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.

18.3. 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 it's 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.

18.4. 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 19. Embedded OpenJPA

19.1. Introduction

Compass has "native" integration with OpenJPA by working in an "embedded" mode within it. OpenJPA can be used with Chapter 18, *JPA (Java Persistence API)* and Compass has specific indexer and lifecycle for it, but Compass can also work from within OpenJPA and have OpenJPA control Compass creation and configuration.

Embedded Compass OpenJPA integration provides support for database indexing and mirroring through the use of the OpenJPA, an implementation of the EJB3 standard.

Compass OpenJPA integration 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 OpenJPA 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).

The Compass OpenJPA uses under the cover Chapter 18, *JPA (Java Persistence API)* and all configuration options apply when using it. The JPA Gps Device extends Compass Gps `AbstractParallelGpsDevice` and supports parallel index operations. It is discussed in more detail here: Section 14.5, “Parallel Device”.

19.2. Configuration

Configuration of Embedded Compass OpenJPA integration is done within the persistence xml file (or programmatic Map configuration) using Compass support for properties based configuration. Here is the most simplest example of enabling Compass within OpenJPA (note, just having Compass jars within the classpath enable it!):

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence persistence_1_0.xsd" version="1.0">

        <persistence-unit name="embeddedopenjpa" transaction-type="RESOURCE_LOCAL">
            <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>
            <class>eg.Test</class>
            <exclude-unlisted-classes>true</exclude-unlisted-classes>
            <properties>
                <property name="openjpa.jdbc.DBDictionary" value="hsql" />
                <property name="openjpa.ConnectionDriverName" value="org.hsqldb.jdbcDriver" />
                <property name="openjpa.ConnectionURL" value="jdbc:hsqldb:mem:test" />
                <property name="openjpa.ConnectionUserName" value="sa" />
                <property name="openjpa.ConnectionPassword" value="" />

                        <!-- This will enable Comapss, this is also the single Compass configuration required --
                <property name="compass.engine.connection" value="target/test-index" />
            </properties>
        </persistence-unit
</persistence>
```

19.3. Index Operation

Embedded Compass OpenJpa provides the ability to index a database (through the use of the JPA device). Indexing the database is simple and can be done using:

```
OpenJPAHelper.getCompassGps(entityManagerFactory).index();
```

Specific configuration for the Compass index instance can be done using `gps.index.compass.` prefix. Internally the `CompassGps` implementation used is `SingleCompassGps`.

Several special properties can also be used. The first, `compass.openjpa.reindexOnStartup` (defaults to `false`) will cause Compass to reindex the database when it starts up. Another important configuration option is `compass.openjpa.indexQuery.[entity name/class]` which allows to plug a custom query string for indexing.

19.4. Real Time Data Mirroring

The embedded Compass OpenJPA integration will automatically register with OpenJPA for lifecycle events and mirror any operation performed using OpenJPA to the database. It also, automatically, integrates with OpenJPA transactions and commits/rollbacks a transaction when OpenJPA transaction commits/rollbacks.

19.5. OpenJPA Helper

`OpenJPAHelper` can be used to obtain the current open `CompassSession` and a `Compass` instance. Both can be obtained from an `EntityManagerFactory` or an `EntityManager`. Prime use case for obtaining a Compass session is to query the index. Note, when querying the index, the returned Objects are not "attached" to JPA, and loaded from the index. This is done for performance reasons as usually they will be used to display results to the user. Attaching them can be done simply by using JPA API.

Chapter 20. Embedded TopLink Essentials

20.1. Introduction

Compass allows for embedded integration with TopLink Essentials. Using simple configuration, Compass will automatically perform mirroring operations (mirroring changes done through TopLink to the search engine), as well as allow to simply index the content of the database using TopLink.

The integration involves few simple steps. The first is enabling Embedded Compass within TopLink. Within the persistence configuration (or when passing properties) a custom Compass TopLink session customizer needs to be defined:

```
<persistence-unit name="test" transaction-type="RESOURCE_LOCAL">
  <provider>oracle.toplink.essentials.PersistenceProvider</provider>
  <properties>
    <!-- ... (other properties) -->
    <property name="toplink.session.customizer"
         value="org.compass.gps.device.jpa.embedded.toplink.CompassSessionCustomizer" />
  </properties>
</persistence-unit>
```

Now that Compass is enabled with TopLink there is one required Compass property in order to configure it which is the location of where the search engine index will be stored. This is configured as a Persistence Unit property configuration using the key `compass.engine.connection` (for example, having the value `file://tmp/index`). When it is configured, Compass will automatically use the mapped TopLink classes and check if one of them is searchable. If there is at least one, then the it will be enabled. That is it!. Now, every operation done using TopLink will be mirrored to the search engine.

Direct access to Compass (for example to execute search operations), can be done using `TopLinkHelper`. For example:

```
Compass compass = TopLinkHelper.getCompass(entityManagerFactory);
CompassSession session = compass.openSession();
CompassTransaction tr = session.beginTransaction();

CompassHits hits = session.find("search something")

tr.commit();
session.close();
```

In order to completely reindex the content of the database based on both the TopLink and Compass mappings, a Compass Gps can be accessed. Here is an example of how to do it:

```
CompassGps gps = TopLinkHelper.getCompassGps(entityManagerFactory);
gps.index();
```

20.2. Configuration

The basic configuration of embedded TopLink Essentials is explained in the introduction section. Within the persistence configuration, the Compass instance used for mirroring and searching can be configured using Compass usual properties (using the `compass.` prefix). If configuring Compass using external configuration is needed, the `compass.toplink.config` can be used to point to Compass configuration file.

The Compass instance created automatically for the indexing operation can be also configured using specific

properties. This properties should have the prefix of `gps.index.`. This is usually configured to have specific parameters for the indexing Compass, for example, having a different index storage location for it while indexing.

20.3. Transaction Management

Compass will integrate with TopLink transaction management (using whichever transaction management it does) by default. If no Compass transaction factory is configured, Compass local transaction factory will be used when using JPA RESOURCE LOCAL transaction type, and JTA sync transaction factory will be used with JPA JTA one.

Chapter 21. Embedded EclipseLink

21.1. Introduction

Compass allows for embedded integration with EclipseLink. Using simple configuration, Compass will automatically perform mirroring operations (mirroring changes done through EcliseLink to the search engine), as well as allow to simply index the content of the database using EclipseLink.

The integration involves few simple steps. The first is enabling Embedded Compass within EclipseLink. Within the persistence configuration (or when passing properties) a custom Compass EclipseLink session customizer needs to be defined:

```
<persistence-unit name="test" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <properties>
    <!-- ... (other properties) -->
    <property name="eclipselink.session.customizer"
         value="org.compass.gps.device.jpa.embedded.eclipselink.CompassSessionCustomizer" />
  </properties>
</persistence-unit>
```

Now that Compass is enabled with EclipseLink there is one required Compass property in order to configure it which is the location of where the search engine index will be stored. This is configured as a Persistence Unit property configuration using the key `compass.engine.connection` (for example, having the value `file://tmp/index`). When it is configured, Compass will automatically use the mapped EclipseLink classes and check if one of them is searchable. If there is at least one, then the it will be enabled. That is it!. Now, every operation done using EclipseLink will be mirrored to the search engine.

Direct access to Compass (for example to execute search operations), can be done using `EclipseLinkHelper`. For example:

```
Compass compass = EclipseLinkHelper.getCompass(entityManagerFactory);
CompassSession session = compass.openSession();
CompassTransaction tr = session.beginTransaction();

CompassHits hits = session.find("search something")

tr.commit();
session.close();
```

In order to completely reindex the content of the database based on both the EclipseLink and Compass mappings, a Compass Gps can be accessed. Here is an example of how to do it:

```
CompassGps gps = EclipseLinkHelper.getCompassGps(entityManagerFactory);
gps.index();
```

21.2. Configuration

The basic configuration of embedded EclipseLink is explained in the introduction section. Within the persistence configuration, the Compass instance used for mirroring and searching can be configured using Compass usual properties (using the `compass.` prefix). If configuring Compass using external configuration is needed, the `compass.eclipselink.config` can be used to point to Compass configuration file.

The Compass instance created automatically for the indexing operation can be also configured using specific

properties. This properties should have the prefix of `gps.index.`. This is usually configured to have specific parameters for the indexing Compass, for example, having a different index storage location for it while indexing.

21.3. Transaction Management

Compass will integrate with EclipseLink transaction management (using whichever transaction management it does) by default. If no Compass transaction factory is configured, Compass local transaction factory will be used when using JPA RESOURCE LOCAL transaction type, and JTA sync transaction factory will be used with JPA JTA one.

Chapter 22. JDO (Java Data Objects)

22.1. 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).

22.2. 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 Jdo `PersistenceManagerFactory`.

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();
```

22.3. 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.

22.4. 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 23. OJB (Object Relational Broker)

23.1. 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).

23.2. 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
<code>index()</code>
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.

23.3. 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.

23.4. 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(ojbDevice, pb);
gps.index();
....
// a Persistence Broker operation level
PersistenceBroker pb = // create Persistence Broker
OjbGpsDeviceUtils.attachPersistenceBrokerForMirror(ojbDevice, pb);
.... // Persistence Broker operations
OjbGpsDeviceUtils.removePersistenceBrokerForMirror(ojbDevice, pb);
```

Chapter 24. iBatis

24.1. 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.

24.2. 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.

24.3. Configuration

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

```
Compass compass = ... // set compass instance
SingleCompassGps 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 25. Introduction

25.1. 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).

25.2. 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.

When using Compass code within an already managed code (within a transaction), it is enough to just use `Compass#openSession()`, without worrying about Compass transaction management code, or even closing the session. Since even opening the session should not be really required, a `LocalCompassSessionBean` can be used to directly inject `CompassSession` to be used. It can be initialized with a `Compass` instance, but if there is only one within Spring application context, it will automatically identify and use it (this feature is similar to @CompassContext annotation explained later).

Compass also supports `@CompassContext` annotations to inject either `Compass` instance or `CompassSession` instance. The annotation can be used on either a class field or on a property setter. In order to inject the annotation, the bean `org.compass.spring.support.CompassContextBeanPostProcessor` need to be added to the bean configuration. If Spring 2 new schema based support is used, `compass:context` can be used.

Compass Spring integration also supports Spring 2 new schema based configuration. Using Compass own schema definition, the configuration of a Compass instance can be embedded within a Spring beans schema based configuration. Here is an example of using the new schema based configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:compass="http://www.compass-project.org/schema/spring-core-config"
     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
       http://www.compass-project.org/schema/spring-core-config http://www.compass-project.org/schema/spring-com

    <bean id="transactionManager" ...>
      ...
    </bean>

    <compass:compass name="compass" txManager="transactionManager">
        <compass:connection>
            <compass:file path="target/test-index" />
        </compass:connection>
    </compass:compass>

    <!-- A direct LocalCompassSessionBean, used with code within a transaciton context -->
    <compass:session id="sess" />
</beans>
```

Chapter 26. DAO Support

26.1. Dao and Template

Compass::Spring uses the `CompassTemplate` and `CompassCallback` classes provided by Compass::Core module as part of it's 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 it's 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 27. Spring Transaction

27.1. 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`.

27.2. 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.

27.3. JTASyncTransaction

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

27.4. 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 transcation 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`.

27.5. CompassTransactionManager

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

Chapter 28. Hibernate 3 Gps Device Support

28.1. Deprecation Note

This device has been deprecated and moved to `org.compass.spring.device.hibernate.dep` package. Please use the new `HibernateGpsDevice` which allows for a pluggable native Hibernate extractor.

28.2. 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.

28.3. 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 29. OJB Gps Device Support

29.1. 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.

29.2. SpringOjbGpsDevice

`SpringOjbGpsDevice` is an extension of the `OjbGpsDevice` and utilizes Spring ojb features. This devices 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).

29.3. 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 30. Jdbc Gps Device Support

30.1. 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);
```

30.2. 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>
```

```
          <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>
```

30.3. 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 31. Spring AOP

31.1. 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.

31.2. 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`).

31.3. 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 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>
```

31.4. Transactional Serivce 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 it's `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 32. Spring MVC Support

32.1. 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.

32.2. 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`.

32.3. 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.

32.4. 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 transcation using the given transaction manager.

Part V. Compass Needle

Compass::Needle provides integration of Compass and Lucene with distributed frameworks. The integration usually revolves around the ability to store the index within a distributed cache, as well as reflecting changes done to the data grid using Compass OSEM into the index.

Chapter 33. GigaSpaces

33.1. Overview

The Compass Needle GigaSpaces integration allows to store a Lucene index within GigaSpaces. It also allows to automatically index the data grid using Compass OSEM support and mirror changes done to the data grid into the search engine.

33.2. Lucene Directory

Compass provides a `GigaSpaceDirectory` which is an implementation of Lucene `Directory` allowing to store the index within GigaSpaces data grid.

Here is a simple example of how it can be used:

```
IJSpace space = SpaceFinder.find("jini://*/*/mySpace");
GigaSpaceDirectory dir = new GigaSpaceDirectory(space, "test");
// ... (use the dir with IndexWriter and IndexSearcher)
```

In the above example we created a directory on top of GigaSpace's Space with an index named "test". The directory can now be used to create Lucene `IndexWriter` and `IndexSearcher`.

The Lucene directory interface represents a virtual file system. Implementing it on top of the Space is done by breaking files into a file header, called `FileEntry` and one or more `FileBucketEntry`. The `FileEntry` holds the meta data of the file, for example, its size and timestamp, while the `FileBucketEntry` holds a bucket size of the actual file content. The bucket size can be controlled when constructing the `GigaSpaceDirectory`, but note that it must not be changed if connecting to an existing index.

Note, it is preferable to configure the directory not to use the compound index format as it yields better performance.

The GigaSpaces integration can also use GigaSpaces just as a distributed lock manager without the need to actually store the index on GigaSpaces. The `GigaSpaceLockFactory` can be used for it.

33.3. Compass Store

Compass allows for simple integration with `GigaSpaceDirectory` as the index storage mechanism. The following example shows how Compass can be configured to work against a GigaSpaces based index with an index named test:

```
<compass name="default">
  <connection>
      <space indexName="test" url="jini://*/*/mySpace"/>
  </connection>
</compass>
```

The following shows how to configure it using properties based configuration:

```
compass.engine.connection=space://test:jini://*/*/mySpace
```

By default, when using GigaSpaces as the Compass store, the index will be in an uncompound file format. It will also automatically be configured with an expiration time based index deletion policy so multiple clients will work correctly.

Compass can also be configured just to used GigaSpaces as a distributed lock manager without the need to actually store the index on GigaSpaces (note that when configuring GigaSpaces as the actual store, the GigaSpaces lock factory will be used by default). Here is how it can be configured:

```
compass.engine.store.lockFactory.type=org.compass.needle.gigaspaces.store.GigaSpaceLockFactoryProvider
compass.engine.store.lockFactory.path=jini://*/*/mySpace?groups=kimchy
```

33.4. Searchable Space

The GigaSpaces integration comes with a built in external data source that can be used with GigaSpaces Mirror Service. Basically, a mirror allows to mirror changes done to the Space (data grid) into the search engine in a reliable asynchronous manner. The following is an example of how it can be configured within a mirror processing unit (for more information see here)

```
<beans xmlns="http://www.springframework.org/schema/beans" ...

  <bean id="compass" class="org.compass.spring.LocalCompassBean">
     <property name="classMappings">
       <list>
         <value>eg.Blog</value>
         <value>eg.Post</value>
         <value>eg.Comment</value>
       </list>
     </property>
     <property name="compassSettings">
       <props>
         <prop key="compass.engine.connection">space://blog:jini://*/*/searchContent</prop>
         <!--  Configure expiration time so other clients that
               haven't refreshed the cache will still see deleted files -->
         <prop key="compass.engine.store.indexDeletionPolicy.type">expirationtime</prop>
         <prop key="compass.engine.store.indexDeletionPolicy.expirationTimeInSeconds">300</prop>
       </props>
     </property>
  </bean>

  <bean id="compassDataSource" class="org.compass.needle.gigaspaces.CompassDataSource">
    <property name="compass" ref="compass" />
  </bean>

  <os-core:space id="mirrodSpace" url="/./mirror-service" schema="mirror"
                 external-data-source="compassDataSource" />
</beans>
```

The above configuration will mirror any changes done in the data grid into the search engine through the Compass instance. It will, further more, connect and store the index content on a specific Space called blog.

Chapter 34. Coherence

34.1. Overview

The Compass Needle Coherence integration allows to store a Lucene index within Coherence Data Grid.

34.2. Lucene Directory

Compass provides two types of Lucene Directory implementations using Coherence, `DataGridCoherenceDirectory` and `InvocableCoherenceDirectory`.

Here is a simple example of how it can be used:

```
CoherenceDirectory dir = new InvocableCoherenceDirectory("cacheName", "indexName");
// ... (use the dir with IndexWriter and IndexSearcher)
dir.close();
```

In the above example we created the invocable Coherence directory on top of Coherence's Data Grid with an index named "test". The directory can now be used to create Lucene `IndexWriter` and `IndexSearcher`.

The Lucene directory interface represents a virtual file system. Implementing it on top of Coherence is done by breaking files into a file header, called `FileEntryKey`/`FileEntryValue` and one or more `FileBucketKey`/`FileBucketValue`. The file header holds the meta data of the file, for example, its size and timestamp, while the file bucket holds a bucket size of the actual file content. The bucket size can be controlled when constructing the coherence directory, but note that it must not be changed if connecting to an existing index.

The `DataGridCoherenceDirectory` uses coherence features that are supported by all of coherence editions. It uses coherence lock API and plain Map remove APIs. The `InvocableCoherenceDirectory` uses coherence invocation service support allowing to delete files (header and buckets) in parallel (without returning results), and use `FileLockKey` existence as an indicator for locking (conditional put) which results in better performance (for remove operations) and better lock API implementation.

Note, it is preferable to configure the directory not to use the compound index format as it yields better performance.

The Coherence integration can also use Coherence just as a distributed lock manager without the need to actually store the index on Coherence. Either the `InvocableCoherenceLockFactory` or `DefaultCoherenceLockFactory` can be used for it.

34.3. Compass Store

Compass allows for simple integration with `DataGridCoherenceDirectory` and `InvocableCoherenceDirectory` as the index storage mechanism. The following example shows how Compass can be configured to work against an invocable coherence directory based index with an index named test and cache name named testcache:

```
<compass name="default">
  <connection>
      <coherence indexName="test" cacheName="testcache"/>
```

```
  </connection>
</compass>
```

The following shows how to configure it using properties based configuration:

```
compass.engine.connection=coherence://test:testcache
```

By default, when using Coherence as the Compass store, the index will be in an uncompound file format. It will also automatically be configured with an expiration time based index deletion policy so multiple clients will work correctly.

Compass can also be configured just to used Coherence as a distributed lock manager without the need to actually store the index on Coherence (note that when configuring Cohernece as the actual store, the Coherence lock factory will be used by default). Here is how it can be configured:

```
compass.engine.store.lockFactory.type=org.compass.needle.cohernece.InvocableCoherenceLockFactoryProvider
compass.engine.store.lockFactory.path=cacheName
```

Chapter 35. Terracotta

35.1. Overview

The Compass Needle Terracotta integration allows to store a Lucene index in a distributed manner using Terracotta as well as provide seamless integration with Compass.

35.2. Lucene Directory

Compass provides a Terracotta optimized directory (similar to Lucene RAM directory) called `TerracottaDirectory`. When using it with pure Lucene applications, the directory needs to be defined as a "root" Terracotta object and then used when constructing `IndexWriter` and `IndexReader`. See the Compass Store on how to use compass jar file as a Terracotta Integration Module (TIM).

Terracotta is a shared memory (referred to as "network attached memory"). The terracotta directory makes use of that and stores the directory in memory allowing for terracotta to distribute changes of it to all relevant nodes connected to the terracotta server. The actual content of a "file" in the directory is broken down into one or more byte arrays, which can be controlled using the bufferSize parameter. Note, once an index is created with a certain bufferSize, it should not be changed. By default, the buffer size is set to `4096` bytes.

Terracotta will automatically fetch required content from the server, and will evict content if memory thresholds break for an application. When constructing large files, the directory allows to set a flush rate when the file content will be flushed (and be allowed to be evicted) during its creation. The formula is that every `bufferSize * flushRate` bytes, it will be released by Compass and allow terracotta to move it to the server and reclaim the memory. The default flush rate is set to 10.

35.3. Compass Store

When using Compass, it is very simple to configure Compass to store the index using Terracotta. Compass jar file already comes in the format of a Terracotta Integration Module (TIM) allowing to simply drop it into `TC_HOME/modules` and it already comes pre-configured with a terracotta configuration of both locks and roots (`terracotta.xml` file within the root of the compass jar file). Another option is to tell Terracotta where to look for more TIMs within the application `tc-config` file and point it to where the compass jar is located.

Once the TIM is setup, Compass has a special Terracotta connection that allows it to use the `TerracottaDirectory` called `TerracottaDirectoryStore`. The `TerracottaDirectoryStore` is where terracotta is configured to have its root (note, this is all defined for you already since compass is a TIM).

Here is a properties/settings based configuration

```
compass.engine.connection=tc://myindex
# default values, just showing how it can be configured
compass.engine.store.tc.bufferSize=4096
compass.engine.store.tc.flushRate=10
```

And here is an xml based configuration:

```
<compass name="default">
  <connection>
      <tc indexName="myindex" bufferSize="4096" flushRate="10" />
  </connection>
```

```
</compass>
```

The "client application" will need to run using Terracotta bootclasspath configuration, and have the following in its `tc-config.xml`:

```
<clients>
    <modules>
        <module group-id="org.compass-project" name="compass" version="2.0.0-RC1" />
    </modules>
</clients>
```

For more information on how to run it in different ways/environments, please refer to the terracotta documentation.

Part VI. Compass Samples

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

Chapter 36. Library Sample

36.1. 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.

36.2. 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 36.1.

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

Chapter 37. Petclinic Sample

37.1. 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).

37.2. 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 37.1.

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.

37.3. 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).

37.3.1. 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 2.0//EN"
    "http://www.compass-project.org/dtd/compass-core-meta-data-2.0.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 birtdate meta-data (in OSEM for example).

37.3.2. 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).

37.3.3. 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.

37.4. 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.

37.4.1. 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-hibernate.xml` in the test package, and the `applicationContext-hibernate.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.

37.4.2. 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.

37.4.3. JDBC

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

on different mappings defentions, 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).

37.5. 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 `serachResource.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.proeprties` 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 VII. 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 connecion string.

Table A.1.

Connection	Description
`file://` prefix or no prefix	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.
`mmap://` prefix	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.
`ram://` prefix	Creates a memory based index, follows the `Compass` life-cycle. Created when the `Compass` is created, and disposed when `Compass` is closed.
`jdbc://` prefix	Holds the Jdbc url or Jndi (based on the `DataSourceProvider` configured). Allows storing the index within a database. This setting requires additional mandatory settings, please refer to the Search Engine Jdbc section. It is very IMPORTANT to read the Search Engine Jdbc 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
compass.name	The name that `Compass` will be registered under. Note that you can specify it at the XML configuration file with a name 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, `Context.INITIAL_CONTEXT_FACTORY`.
compass.jndi.url	JNDI provider URL, `Context.PROVIDER_URL`
compass.jndi.*	prefix for arbitrary JNDI `InitialContext` 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 `alias` (set it only if one of your mapped meta data is called alias).
compass.property.extendedAlias	The name of the property that extended aliased (if exists) of a given Resource will be stored. This allows for poly alias queries where one can query on a "base" alias, and get all the aliases the are extending it. Defaults to `extendedAlias` (set it only if one of your mapped meta data is called extendedAlias).
compass.property.all	The name of the "all" property that Compass will use (a property that accumulates all the properties/meta-data). Defaults to `all` (set it only if one of your mapped meta data is called all). Note that it can be overriden 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 `no`, `yes`, `positions`, `offsets`, or `positions_offsets`.

A.1.4. Transaction Level

Compass supports several transaction isolation levels. More information about them can be found in the Search Engine chapter.

Table A.4.

Transaction Level	Description
none	Not supported, upgraded to `read_committed`.

Transaction Level	Description
read_uncommitted	Not supported, upgraded to `read_committed`.
read_committed	The same read committed from data base systems. As fast for read only transactions.
repeatable_read	Not supported, upgraded to `serializable`.
serializable	The same as serializable from data base systems. Performance killer, basically results in transactions executed sequentially.
lucene (batch_insert)	A special transaction level, `lucene` (previously known as `batch_insert`) isolation level is similar to the `read_committed` isolation level except dirty operations done during a transaction are not visible to get/load/find operations that occur within the same transaction. This isolation level is very handy for long running batch dirty operations and can be faster than `read_committed`. Most usage patterns of Compass (such as integration with ORM tools) can work perfectly well with the `lucene` isolation level.

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
org.compass.core.transaction.LocalTransactionFactory	Manages a local transaction which does not interact with other transaction mechanisms.
org.compass.core.transaction.JTASyncTransactionFactory	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.
org.compass.core.transaction.XATransactionFactory	Uses the JTA Transaction to enlist a Compass implemented `XAResource` This allows for Compass to participate in a two phase commit operation. Note, the JTA implementation should automatically delist the resource when the transaction commit/rollback. 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.JOnaAS	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
compass.propertyAccessor.[property accessor name].type	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 it's 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 type	Lookup Name	Converter Class	Converter Type	Notes
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.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 `InputStream` 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	
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
compass.converter.[converter name].type	The type of the `org.compass.converter.Converter` 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).
compass.converter.[converter name].format	Applies to format-able converters. The format that will be used to format the data converted (see Java `java.text.DecimalFormat` and `java.text.SimpleDateFormat`).
compass.converter.[converter name].format.locale	The `Locale` to be used when formatting.
compass.converter.[converter name].format.minPoolSize	Compass pools the formatters for greater performance. The value of the minimum pool size. Defaults to 4.
compass.converter.[converter	Compass pools the formatters for greater performance. The value of

Setting	Description
name].format.maxPoolSize	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 than 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
compass.engine.connection	The index engine file system location.
compass.engine.defaultsearch	When searching using a query string, the default property/meta-data that compass will use for non prefixed strings. Defaults to `compass.property.all` value.
compass.engine.all.analyzer	The name of the analyzer to use for the all property (see the next section about Search Engine Analyzers).
compass.transaction.lockDir	The directory where the search engine will maintain it's locking file mechanism for inter and outer process transaction synchronization. Defaults to the `java.io.tmpdir` Java system property. This is a JVM level property.
compass.transaction.lockTimeout	The amount of time a transaction will wait in order to obtain it's specific lock (in seconds). Defaults to 10 seconds.
compass.transaction.lockPollInterval	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 proeprty.
compass.engine.optimizer.type	The fully qualified class name of the search engine optimizer that will be used. Defaults to `org.compass.core.lucene.engine.optimizer.AdaptiveOptimizer`. Please see the following section for a list of optimizers.
compass.engine.optimizer.schedule	Determines if the optimizer will be scheduled or not (`true` or `false`), defaults to `true`. 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.fixedRate	Determines if the schedule will run in a fixed rate or not. If it is set to `false` each execution is scheduled relative to the actual execution of the previous execution. If it is set to `true` 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	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	Determines the minimal number of documents required before the buffered in-memory documents are flushed as a new Segment. Large values generally gives faster indexing. When this is set, the writer will flush every maxBufferedDocs added documents. Pass in -1 to prevent triggering a flush due to number of buffered documents. Note that if flushing by RAM usage is also enabled, then the flush will be triggered by whichever comes first. Disabled by default (writer flushes by RAM usage).
compass.engine.maxBufferedDeletedTerms	Determines the minimal number of delete terms required before the buffered in-memory delete terms are applied and flushed. If there are documents buffered in memory at the time, they are merged and a new segment is created. Disabled by default (writer flushes by RAM usage).
compass.engine.ramBufferSize	Determines the amount of RAM that may be used for buffering added documents before they are flushed as a new Segment. Generally for faster indexing performance it's best to flush by RAM usage instead of document count and use as large a RAM buffer as you can. When this is set, the writer will flush whenever buffered documents use this much RAM. Pass in -1 to prevent triggering a flush due to RAM usage. Note that if flushing by document count is also enabled, then the flush will be triggered by whichever comes first. The default value is 16 (M).
compass.engine.termIndexInterval	Expert: Set the interval between indexed terms. Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an

Setting	Description
	IndexReader, and speed random-access to terms. This parameter determines the amount of computation required per query term, regardless of the number of documents that contain that term. In particular, it is the maximum number of other terms that must be scanned before a term is located and its frequency and position information may be processed. In a large index with user-entered query terms, query processing time is likely to be dominated not by term lookup but rather by the processing of frequency and positional data. In a small index or when many uncommon query terms are generated (e.g., by wildcard queries) term lookup may become a dominant cost. In particular, numUniqueTerms/interval terms are read into memory by an IndexReader, and, on average, interval/2 terms must be scanned for each random term access.
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 (`true`) or off (`false`) the use of compound files. If used lowers the number of files open, but have very small performance overhead. Defaults to `true`. 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 a single instance is working with the index, since the cache is automatically invalidated upon a dirty operation.
compass.engine.indexManagerScheduleInterval	The index manager schedule interval (in seconds) where different actions related to index manager will happen (such as global cache interval invalidation checks - see `SearchEngineIndexManager#notifyAllToClearCache` and `SearchEngineIndexManager#checkAndClearIfNotifiedAllToClearCache`). Defaults to 60 seconds.
compass.engine.waitForCacheInvalidationOnIndexOperation	Defaults to false. 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 indexManagerScheduledInterval 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
`org.compass.core.lucene.engine.optimizer.AdaptiveOptimizer`	When the number of segments exceeds that specified `mergeFactor`, the segments will be merged from the last segment, until a segment with a higher resource count will be encountered.
`org.compass.core.lucene.engine.optimizer.AggressiveOptimizer`	When the number of segments exceeds that specified `mergeFactor`, all the segments are merged into a single segment.
`org.compass.core.lucene.engine.optimizer.NullOptimizer`	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
compass.engine.store.jdbc. dialect	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 not set, Compass will try to auto-detect it based on the Database meta-data.
compass.engine.store.jdbc. disableSchemaOperations	Optional. If set to `true`, 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 `false`.
compass.engine.store.jdbc. managed	Optional (defaults to `false`). If the connection is managed or not. Basically, if set to `false`, compass will commit and rollback the transaction. If set to `true`, compass will not perform it. Defaults to `false`. Should be set to `true` if using external transaction managers (like JTA or Spring `PlatformTransactionManager`), and `false` if using compass `LocalTransactionFactory`. Note as well, that if using external transaction managers, the `compass.transaction.commitBeforeCompletion` should be set to `true`. If the connection is not managed (set to `false`), the created `DataSource` will be wrapped with Compass Jdbc directory `TransactionAwareDataSourceProxy`. Please refer to Lucene Jdbc Directory appendix for more information.

Setting	Description
compass.engine.store.jdbc. connection.provider.class	The fully qualified name of the `DataSourceProvider`. The `DataSourceProvider` is responsible for getting/creating the Jdbc `DataSource` that will be used. Defaults to `org.compass.core.lucene.engine .store.jdbc.DriverManagerDataSourceProvider` (Poor performance). Please refer to next section for a list of the available providers.
compass.engine.store.jdbc. useCommitLocks	Optional (defaults to `false`). Determines if the index will use Lucene commit locks. Setting it to `true` makes sense only if the system will work in `autoCommit` 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 `PhantomReadLock`). The fully qualified name of the `Lock` implementation that will be used.
compass.engine.store.jdbc. ddl.name.name	Optional (defaults to `name_`). 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 `value_`). 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. ddl.size.name	Optional (defaults to `size_`). The name of the size column.
compass.engine.store.jdbc. ddl.lastModified.name	Optional (defaults to `lf_`). The name of the last modified column.
compass.engine.store.jdbc. ddl.deleted.name	Optional (defaults to `deleted_`). The name of the deleted column.

A.1.9.1. Data Source Providers

Compass comes with several built in `DataSourceProviders`. 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
DriverManagerDataSourceProvider	The default data source provider. Creates a simple `DataSource` that returns a new `Connection` for each request. Performs very poorly,

Data Source Provider Class	Description
	and should not be used.
DbcpDataSourceProvider	Uses Jakarta Commons DBCP Connection pool. Compass provdes several additional configurations settings to configure DBCP, please refer to `LuceneEnvironment#DataSourceProvider#Dbcp` javadoc.
C3P0DataSourceProvider	Uses C3P0 Connection pool. Configring additional properties for the C3P0 connection pool uses C3p0 internal support for a `c3p0.properties` that should reside as a top-level resource in the same CLASSPATH / classloader that loads c3p0's jar file.
JndiDataSourceProvider	Gets a `DataSource` from JNDI. The JNDI name is the value after the `jdbc://` prefix in Compass connection setting.
ExternalDataSourceProvider	A data source provider that can use an externally configured data source. In order to set the external DataSource to be used, the `ExternalDataSourceProvider#setDataSource(DataSource)` static method needs to be called before the `Compass` instance if 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 Jdbc driver class.
compass.engine.store.jdbc. connection.username	The Jdbc connection user name.
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 `FetchPerTransactoinJdbcIndexInput` 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 `IndexInput` implementation.
compass.engine.store.jdbc.fe.[name].indexOutput.type	The fully qualified class name of the `IndexInput` 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 `IndexInput` 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 `IndexOutput` implementations.
compass.engine.store.jdbc.fe.[name].indexOutput.threshold	The threshold value (in bytes) after which data will be temporarly written to a file (and them dumped into the database). Applies when using `RAMAndFileJdbcIndexOutput` (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
compass.engine.analyzer.[analyzer name].type	The type of the search engine analyzer, please see the available analyzers types later in the section.
compass.engine.analyzer.[analyzer name].filters	A comma separated list of `LuceneAnalyzerTokenFilterProviders` registered under compass, to be applied for the given analyzer. For example, adding a synonym analyzer, you should register a synonym `LuceneAnalyzerTokenFilterProvider` under your own choice for filter name, and add it to the list of filters here.
compass.engine.analyzer.[analyzer name].stopwords	A comma separated list of stop words to use with the chosen analyzer. If the string starts with `+`, the list of stop-words will be added to the default set of stop words defined for the analyzer.

Setting	Description
	Note, that not all the analyzers type support this feature.
compass.engine.analyzer.[analyzer name].factory	If the `compass.engine.analyzer.[analyzer name].type` setting is not enough to configure your analyzer, use it to define the fully qualified class name of your analyzer factory which implements `LuceneAnalyzerFactory` 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 analyer types comes with the `lucene-analyzers` jar. They are: `brazilian`, `cjk`, `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 `org.compass.core.lucene.engine.analyzer.LuceneAnalyzerTokenFilterPr` interface. Can also be the value `synonym`, which will automatically map to the `org.compass.core.lucene.engine.analyzer.synonym.SynonymAnalyzerToke` class.
compass.engine.analyzerfilter.[analyzer filter name].lookup	Only applies for synonym filters. The class that implements the `org.compass.core.lucene.engine.analyzer.synonym.SynonymLookupProvid` 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 than 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 `DefaultLuceneHighlighterFactory`). The fully qualified name of the class that creates highlighters settings. Must implement the `LuceneHighlighterFactory` interface.
compass.engine.highlighter.[highlighter name].textTokenizer	Optional (default to `auto`). Defines how a text will be tokenized to be highlighted. Can be `analyzer` (use an analyzer to tokenize the text), `term_vector` (use the term vector info stored in the index), or `auto` (will first try `term_vector`, and if no info is stored, will try to use `analyzer`).
compass.engine.highlighter.[highlighter name].rewriteQuery	Low level. Optional (defaults to `true`). 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 `...`). Sets the separator string between fragments if using the combined fragments highlight option.
compass.engine.highlighter.[highlighter name].maxBytesToAnalyze	Optional (defaults to `50*1024`). Sets the maximum byes of text to analyze.
compass.engine.highlighter.[highlighter name].fragmenter.type	Optional (default to `simple`). The type of the fragmenter that will be used, can be `simple`, `null`, or the fully qualified class name of the fragmenter (implements the `org.apache.lucene.search.highlight.Fragmenter`).
compass.engine.highlighter.[highlighter name].fragmenter.simple.size	Optional (defaults to `100`). Sets the size (in bytes) of the fragments for the `simple` fragmenter.
compass.engine.highlighter.[highlighter name].encoder.type	Optional (default to `default`). The type of the encoder that will be used to encode fragmented text. Can be `default` (does nothing), `html` (escapes html tags), or the fully qualifed class name of the encoder (implements `org.apache.lucene.search.highlight.Encoder`).
compass.engine.highlighter.[highlighter name].formatter.type	Optional (default to `simple`). The type of the formatter that will be used to highlight the text. Can be `simple` (simply wraps the highlighted text with pre and post strings), `htmlSpanGradient` (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 formatter (implements `org.apache.lucene.search.highlight.Formatter`).
compass.engine.highlighter.[highlighter name].formatter.simple.pre	Optional (default to `<b>`). In case the highlighter uses the `simple` formatter, controlls the text that is appened before the highlighted text.

Setting	Description
compass.engine.highlighter.[highlighter name].formatter.simple.post	Optional (default to `</b>`). In case the highlighter uses the `simple` formatter, controlls the text that is appened after the highlighted text.
compass.engine.highlighter.[highlighter name].formatter.htmlSpanGradient.maxScore	In case the highlighter uses the `htmlSpanGradient` 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 `htmlSpanGradient` formatter, the hex color used for representing IDF scores of zero eg #FFFFFF (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 `htmlSpanGradient` formatter, the largest hex color used for representing IDF scores eg #000000 (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 `htmlSpanGradient` formatter, the hex color used for representing IDF scores of zero eg #FFFFFF (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 `htmlSpanGradient` formatter, The largest hex color used for representing IDF scores eg #000000 (black).

A.1.13. Other Settings

Several other settings that control compass.

Table A.20.

Setting	Description
compass.osem.managedId.index	Can be either `un_tokenized` or `no` (defaults to `no`). 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 `un_tokenized`).
compass.osem.supportUnmarshall	Controls if the default support for un-marshalling within the class mappings will default to `true` or `false` (unless it is explicitly set in the class mapping). Defaults to `true`. Controls if the searchable class will support unmarshalling from the search engine or using `Resource` is enough. Un-marshalling is the process of converting a raw `Resource` into the actual domain object. If support un-marshall is enabled extra information will be stored within the search engine, as well as consumes extra memory

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

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 `Blob` 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 `JdbcDirectorySettings`, or provided as a parameter to the deleteMarkDeleted 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*
`org.apache.lucene.store.jdbc.dialect.OracleDialect`	Oracle	Oracle Jdbc Driver - Yes
`org.apache.lucene.store.jdbc.dialect.SQLServerDialect`	Microsoft SQL Server	jTds 1.2 - No. Microsoft Jdbc Driver - Unknown
`org.apache.lucene.store.jdbc.dialect.MySQLDialect`	MySQL	MySQL Connector J 3.1/5 - Yes with `emulateLocators=true` in connection string.
`org.apache.lucene.store.jdbc.dialect.MySQLInnoDBDialect`	MySQL with InnoDB.	See MySQL
`org.apache.lucene.store.jdbc.dialect.MySQLMyISAMDialect`	MySQL with MyISAM	See MySQL
`org.apache.lucene.store.jdbc.dialect.PostgreSQLDialect`	PostgreSQL	Postgres Jdbc Driver - Yes.
`org.apache.lucene.store.jdbc.dialect.SybaseDialect`	Sybase/ Sybase Anywhere	Unknown.
`org.apache.lucene.store.jdbc.dialect.InterbaseDialect`	Interbase	Unknown.
`org.apache.lucene.store.jdbc.dialect.FirebirdDialect`	Firebird	Unknown.

Dialect	RDBMS	Blob Locator Support*
`org.apache.lucene.store.jdbc.dialect.DB2Dialect`	DB2 / DB2 AS400 / DB2 OS390	Unknown.
`org.apache.lucene.store.jdbc.dialect.DerbyDialect`	Derby	Derby Jdbc Driver- Unknown.
`org.apache.lucene.store.jdbc.dialect.HSQLDialect`	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 uncompund 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 `Connection`s.

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;
} finnaly {
    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. IMORTANT: 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 `FileEntryHandlers`. They are:

Table B.5. File Entry Handler Types

Type	Description
`org.apache.lucene.store.jdbc.handler.NoOpFileEntryHandler`	Performs no operations.
`org.apache.lucene.store.jdbc.handler.ActualDeleteFileEntryHandler`	Performs actual delete from the database when the different delete operations are called. Also support configurable `IndexInput` 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 `IndexInput` is opened. Perfect for small sized file entries (like the segments file).
`org.apache.lucene.store.jdbc.index.FetchOnBufferReadJdbcIndexInput`	Extends the `JdbcBufferedIndexInput` class, and fetches the data from the database every time the internal buffer need to be refilled. The `JdbcBufferedIndexInput` allows setting the buffer size using the `JdbcBufferedIndexInput#BUFFER_SIZE_SETTING`. Remember, that you can set different buffer size for different files by registering different file entry handlers with the `JdbcDirectorySettings`.
`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 `FetchPerTransactionJdbcIndexInput#releaseBlobs(java.sql.Connection)` when the transaction ends. It is automatically taken care of if using `TransactionAwareDataSourceProxy`. If using JTA for example, a transcation synchronization should be registered with JTA to clear the blobs. Extends the `JdbcBufferedIndexInput` class, and fetches the data from the database every time the internal buffer need to be

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

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
`org.apache.lucene.store.jdbc.index.RAMJdbcIndexOutput`	Extends the `JdbcBufferedIndexOutput` class, and stores the data to be written in memory (within a growing list of bufferSize sized byte arrays). The `JdbcBufferedIndexOutput` allows setting the buffer size using the `JdbcBufferedIndexOutput#BUFFER_SIZE_SETTING`. Perfect for small sized file entries (like the segments file).
`org.apache.lucene.store.jdbc.index.FileJdbcIndexOutput`	Extends the `JdbcBufferedIndexOutput` class, and stores the data to be written in a temporary file. The `JdbcBufferedIndexOutput` allows setting the buffer size using the `JdbcBufferedIndexOutput#BUFFER_SIZE_SETTING` (a write is performed every time the buffer is flushed).
`org.apache.lucene.store.jdbc.index.RAMAndFileJdbcIndexOutput`	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 cab be configured using `RAMAndFileJdbcIndexOutput#INDEX_OUTPUT_THRESHOLD_SETTING`.

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.