Hibernate OGM Reference Guide

4.1.3.Final

by Emmanuel Bernard (Red Hat), Sanne Grinovero (Red Hat), Gunnar Morling (Red Hat), and Davide D'Alto (Red Hat)

Preface vii
1. Goals viii
2. What we have today ix
3. Experimental features x
4. Use cases x
1. How to get help and contribute on Hibernate OGM 1
1.1. How to get help 1
1.2. How to contribute 1
1.2.1. How to build Hibernate OGM 1
1.2.2. How to contribute code effectively 2
2. Getting started with Hibernate OGM 5
3. Architecture 11
3.1. General architecture 11
3.2. How is data persisted 13
3.3. How is data queried 19
4. Configure and start Hibernate OGM 23
4.1. Bootstrapping Hibernate OGM 23
4.1.1. Using JPA 23
4.1.2. Using Hibernate ORM native APIs 24
4.2. Environments 25
4.2.1. In a Java EE container 25
4.2.2. In a standalone JTA environment 26
4.2.3. Without JTA 27
4.3. Configuration options 28
4.4. Configuring Hibernate Search 28
4.5. How to package Hibernate OGM applications for WildFly 8.2 28
4.5.1. Packaging Hibernate OGM applications for WildFly 8.2 29
4.5.2. Loading both the Hibernate Search and Hibernate OGM modules WildFly 8.2 30
5. Map your entities 31
5.1. Supported entity mapping 31
5.2. Supported Types 32
5.3. Supported association mapping 32
6. Hibernate OGM APIs 35
6.1. Bootstrap Hibernate OGM 35
6.2. JPA and native Hibernate ORM APIs 35
6.2.1. Accessing the OgmSession API 36
6.3. On flush and transactions 36
6.4. SPIs 37
7. Query your entities 39
7.1. Using JP-QL 39
7.2. Using the native query language of your NoSQL 42
7.3. Using Hibernate Search 43
7.4. Using the Criteria API 44

8. NoSQL datastores 45
8.1. Using a specific NoSQL datastore 45
9. Infinispan 47
9.1. Configure Infinispan 47
9.1.1. Adding Infinispan dependencies 47
9.1.2. Infinispan specific configuration properties 48
9.1.3. Cache names used by Hibernate OGM 49
9.2. Manage data size 50
9.3. Clustering: store data on multiple Infinispan nodes 51
9.4. Storage principles 54
9.4.1. Properties and built-in types 54
9.4.2. Identifiers 55
9.4.3. Entities 59
9.4.4. Associations 63
9.5. Transactions 76
9.6. Storing a Lucene index in Infinispan 77
10. Ehcache 79
10.1. Configure Ehcache 79
10.1.1. Adding Ehcache dependencies 79
10.1.2. Ehcache specific configuration properties 79
10.1.3. Cache names used by Hibernate OGM 80
10.2. Storage principles 81
10.2.1. Properties and built-in types 81
10.2.2. Identifiers 82
10.2.3. Entities 86
10.2.4. Associations 90
10.3. Transactions 103
11. MongoDB 105
11.1. Configuring MongoDB 105
11.1.1. Adding MongoDB dependencies 105
11.1.2. MongoDB specific configuration properties 105
11.1.3. Annotation based configuration 108
11.1.4. Programmatic configuration 109
11.2. Storage principles 110
11.2.1. Properties and built-in types 110
11.2.2. Entities 113
11.2.3. Associations 126
11.3. Transactions 155
11.4. Optimistic Locking 155
11.5. Queries 156
11.5.1. JP-QL queries 157
11.5.2. Native MongoDB queries 157
11.5.3. Hibernate Search 160
12. Neo4j 161

12.1. How to add Neo4j integration 161
12.2. Configuring Neo4j 161
12.3. Storage principles 162
12.3.1. Properties and built-in types 163
12.3.2. Entities 164
12.3.3. Associations 169
12.3.4. Auto-generated Values 178
12.3.5. Labels summary 180
12.4. Transactions 181
12.5. Queries 181
12.5.1. JP-QL queries 182
12.5.2. Cypher queries 182
13. CouchDB (Experimental) 185
13.1. Configuring CouchDB 185
13.1.1. Annotation based configuration 186
13.1.2. Programmatic configuration 187
13.2. Storage principles 188
13.2.1. Properties and built-in types 188
13.2.2. Entities 190
13.2.3. Associations 200
13.3. Transactions 222
13.4. Queries 222

Preface

Hibernate Object/Grid Mapper (OGM) is a persistence engine providing Java Persistence (JPA) support for NoSQL datastores. It reuses Hibernate ORM’s object life cycle management and (de)hydration engine but persists entities into a NoSQL store (key/value, document, column-oriented, etc) instead of a relational database. It reuses the Java Persistence Query Language (JP-QL) as an interface to querying stored data.

The project is now fairly mature when it comes to the storage strategies. And the feature set is sufficient to be used in your projects. We do have however much bigger ambitions than a simple object mapper. Many things are on the roadmap (more NoSQL, query, denormalization engine, etc). If you feel a feature is missing, *report it to us*. If you want to contribute, *even better*!

Hibernate OGM is released under the LGPL open source license.

Note

The future of this project is being shaped by the requests from our users. Please give us feedback on

- what you like
- what you don’t like
- what is confusing
- what you are missing as a feature

Check *Section 1.2, “How to contribute”* on how to contact us.

Tip

We worked hard on this documentation but we know it is far from perfect. If you find something confusing or feel that an explanation is missing, contact us in one of the following ways:

- post on *Hibernate OGM’s forum* [https://forum.hibernate.org/viewforum.php?f=31].
- open bug reports in *JIRA* [https://hibernate.atlassian.net/browse/OGM]
- propose improvements on the *development mailing list* [http://www.hibernate.org/community/mailinglists]

- join us on IRC to discuss developments and improvements (`#hibernate-dev` on `freenode.net`; you need to be registered on freenode: the room does not accept "anonymous" users).

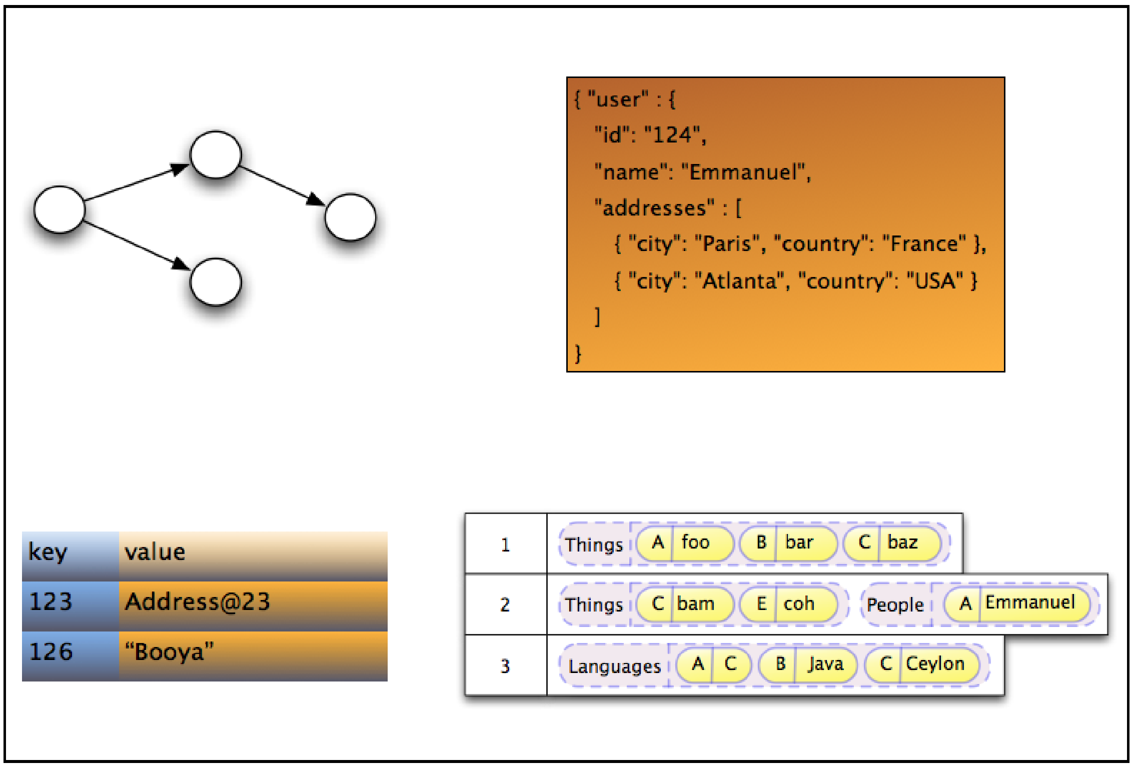
1. Goals

Hibernate OGM:

- offers a familiar programming paradigm (JPA) to deal with NoSQL stores
- moves model denormalization from a manual imperative work to a declarative approach handled by the engine
- encourages new data usage patterns and NoSQL exploration in more "traditional" enterprises
- helps scale existing applications with a NoSQL front end to a traditional database

NoSQL can be very disconcerting as it is composed of many disparate solutions with different benefits and drawbacks. Speaking only of the main ones, NoSQL is at least categorized in four families:

- graph oriented databases
- key/value stores: essentially Maps but with different behaviors and ideas behind various products (data grids, persistent with strong or eventual consistency, etc)
- document based datastores: contains as value semi-structured documents (think JSON)
- column based datastores



```
{ "user" : {
   "id": "124",
   "name": "Emmanuel",
   "addresses" : [
      { "city": "Paris", "country": "France" },
      { "city": "Atlanta", "country": "USA" }
   ]
}
```

key	value
123	Address@23
126	"Booya"

Figure 1. Various NoSQL families

Each have different benefits and drawbacks and one solution might fit a use case better than an other. However access patterns and APIs are different from one product to the other.

Hibernate OGM is not expected to be the Rosetta stone used to interact with *all* NoSQL solution in *all* use cases. But for people modeling their data as a domain model, it provides distinctive advantages over raw APIs and has the benefit of providing an API and semantic known to Java developers. Reusing the same programmatic model and trying different (No)SQL engines will hopefully help people to explore alternative datastores.

Hibernate OGM also aims at helping people scale traditional relational databases by providing a NoSQL front-end and keeping the same JPA APIs and domain model.

2. What we have today

Today, Hibernate OGM does not support all of these goals. Here is a list of what we have:

- store data in key/value stores (Infinispan's datagrid and Ehcache)
- store data in document stores (MongoDB and CouchDB - the latter in preview)
- store data in graph databases (Neo4J)

- Create, Read, Update and Delete operations (CRUD) for entities
- polymorphic entities (support for superclasses, subclasses etc).
- embeddable objects (aka components)
- support for basic types (numbers, String, URL, Date, enums, etc)
- support for associations
- support for collections (Set, List, Map, etc)
- support for JP-QL queries (not arbitrary joins though)
- support for mapping native queries results to managed entities
- support for Hibernate Search's full-text queries
- and generally, support for JPA and native Hibernate ORM API support

In short, a perfectly capable Object Mapper for multiple popular NoSQL datastores.

3. Experimental features

As Hibernate OGM is a rather young project, some parts of it may be marked as experimental. This may affect specific APIs or SPIs (e.g. the case for the `SchemaInitializer` SPI contract at the moment), entire dialects (this is the case for the CouchDB dialect at the moment) or deliverables.

Experimental APIs/SPIs are marked via the `@Experimental` annotation. Experimental dialects make that fact apparent through their datastore name (e.g. "COUCHDB_EXPERIMENTAL") and experimental deliverables use the "experimental" artifact classifier.

If a certain part is marked as experimental it may undergo backwards-incompatible changes in future releases. E.g. API/SPI methods may be altered, so that code using them needs to be adapted as well. For experimental dialects the persistent format of data may be changed, so that a future version of such dialect may not be able to read back data written by previous versions. A manual update of the affected data may be thus required. Experimental deliverables should be used with special care, as they are work in progress. You should use them for testing but not production use cases.

But most of our dialects are mature, so don't worry ;)

4. Use cases

Here are a few areas where Hibernate OGM can be beneficial:

- need to scale your datastore up and down rapidly (via the underlying NoSQL datastore capability)

- keep your domain model independent of the underlying datastore technology (RDBMS, Infinispan, NoSQL)
- explore the best tool for the use case
- use a familiar JPA front end to your datastore
- use Hibernate Search full-text search / text analysis capabilities and store the data set in an scalable datastore

These are a few ideas and the list will grow as we add more capabilities to Hibernate OGM.

How to get help and contribute on Hibernate OGM

Hibernate OGM is a young project. Join and help us shape it!

1.1. How to get help

First of all, make sure to read this reference documentation. This is the most comprehensive formal source of information. Of course, it is not perfect: feel free to come and ask for help, comment or propose improvements in our *Hibernate OGM forum* [https://forum.hibernate.org/viewforum.php?f=31].

You can also:

- open bug reports in *JIRA* [https://hibernate.atlassian.net/browse/OGM]
- propose improvements on the *development mailing list* [http://www.hibernate.org/community/mailinglists]
- join us on IRC to discuss developments and improvements (`#hibernate-dev` on `freenode.net`; you need to be registered on freenode: the room does not accept "anonymous" users).

1.2. How to contribute

Welcome!

There are many ways to contribute:

- report bugs in *JIRA* [https://hibernate.atlassian.net/browse/OGM]
- give feedback in the forum, IRC or the development mailing list
- improve the documentation
- fix bugs or contribute new features
- propose and code a datastore dialect for your favorite NoSQL engine

Hibernate OGM’s code is available on GitHub at *https://github.com/hibernate/hibernate-ogm*.

1.2.1. How to build Hibernate OGM

Hibernate OGM uses Git and Maven 3, make sure to have both installed on your system.

Clone the git repository from GitHub:

```
#get the sources
git clone https://github.com/hibernate/hibernate-ogm
cd hibernate-ogm
```

Run maven

```
#build project
mvn clean install -s settings-example.xml
```

Note

Note that Hibernate OGM uses artifacts from the Maven repository hosted by JBoss. Make sure to either use the `-s settings-example.xml` option or adjust your `~/.m2/settings.xml` according to the descriptions available *on this jboss.org wiki page* [http://community.jboss.org/wiki/MavenGettingStarted-Users].

To skip building the documentation, set the `skipDocs` property to true:

```
mvn clean install -DskipDocs=true -s settings-example.xml
```

Tip

If you just want to build the documentation only, run it from the `hibernate-ogm-documentation/manual` subdirectory.

1.2.2. How to contribute code effectively

The best way to share code is to fork the Hibernate OGM repository on GitHub, create a branch and open a pull request when you are ready. Make sure to rebase your pull request on the latest version of the master branch before offering it.

Here are a couple of approaches the team follows:

- We do small independent commits for each code change. In particular, we do not mix stylistic code changes (import, typos, etc) and new features in the same commit.
- Commit messages follow this convention: the JIRA issue number, a short commit summary, an empty line, a longer description if needed. Make sure to limit line length to 80 characters, even at this day and age it makes for more readable commit comments.

```
OGM-123 Summary of commit operation

```

```
Optional details on the commit
and a longer description can be
added here.
```

- A pull request can contain several commits but should be self contained: include the implementation, its unit tests, its documentation and javadoc changes if needed.
- All commits are proposed via pull requests and reviewed by another member of the team before being pushed to the reference repository. That’s right, we never commit directly upstream without code review.

Getting started with Hibernate OGM

If you are familiar with JPA, you are almost good to go :-) We will nevertheless walk you through the first few steps of persisting and retrieving an entity using Hibernate OGM.

Before we can start, make sure you have the following tools configured:

- Java JDK 6 or above
- Maven 3.x

Hibernate OGM is published in the JBoss hosted Maven repository. Adjust your `~/.m2/settings.xml` file according to the guidelines found *on this webpage* [http://community.jboss.org/wiki/MavenGettingStarted-Users]. In this example we will use Infinispan as the targeted datastore.

Add `org.hibernate.ogm:hibernate-ogm-bom:4.1.3.Final` to your dependency management block and `org.hibernate.ogm:hibernate-ogm-infinispan:4.1.3.Final` to your project dependencies:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.hibernate.ogm</groupId>
            <artifactId>hibernate-ogm-bom</artifactId>
            <version>4.1.3.Final</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
<dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.hibernate.ogm</groupId>
        <artifactId>hibernate-ogm-infinispan</artifactId>
    </dependency>
</dependencies>
```

The former is a so-called "bill of materials" POM which specifies a matching set of versions for Hibernate OGM and its dependencies. That way you never need to specify a version explicitly within your dependencies block, you will rather get the versions from the BOM automatically.

Note

If you're deploying your application onto JBoss WildFly, you don't need to add the Hibernate OGM modules to your deployment unit but you can rather add them as modules to the application server itself. Refer to *Section 4.5, "How to package Hibernate OGM applications for WildFly 8.2"* to learn more.

We will use the JPA APIs in this tutorial. While Hibernate OGM depends on JPA 2.1, it is marked as provided in the Maven POM file. If you run outside a Java EE container, make sure to explicitly add the dependency:

```
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
</dependency>
```

Let’s now map our first Hibernate OGM entity.

```
@Entity
public class Dog {
   @Id @GeneratedValue(strategy = GenerationType.TABLE, generator = "dog")
   @TableGenerator(
      name = "dog",
      table = "sequences",
      pkColumnName = "key",
      pkColumnValue = "dog",
      valueColumnName = "seed"
   )
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   private Long id;

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

   @ManyToOne
   public Breed getBreed() { return breed; }
   public void setBreed(Breed breed) { this.breed = breed; }
   private Breed breed;
}

@Entity
public class Breed {

   @Id @GeneratedValue(generator = "uuid")
   @GenericGenerator(name="uuid", strategy="uuid2")
   public String getId() { return id; }
   public void setId(String id) { this.id = id; }
   private String id;

   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
   private String name;
}
```

I lied to you, we have already mapped two entities! If you are familiar with JPA, you can see that there is nothing specific to Hibernate OGM in our mapping.

In this tutorial, we will use JBoss Transactions for our JTA transaction manager. So let's add the JTA API and JBoss Transactions to our POM as well. The final list of dependencies should look like this:

```
<dependencies>
    <!-- Hibernate OGM Infinispan module; pulls in the OGM core module -->
    <dependency>
        <groupId>org.hibernate.ogm</groupId>
        <artifactId>hibernate-ogm-infinispan</artifactId>
    </dependency>

    <!-- standard APIs dependencies - provided in a Java EE container -->
    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.1-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.transaction</groupId>
        <artifactId>jboss-transaction-api_1.2_spec</artifactId>
    </dependency>

    <!-- JBoss Transactions dependency - this (or another implementation) is
         provided in a Java EE container -->
    <dependency>
        <groupId>org.jboss.jbossts</groupId>
        <artifactId>jbossjta</artifactId>
    </dependency>
</dependencies>
```

Next we need to define the persistence unit. Create a `META-INF/persistence.xml` file.

```
<?xml version="1.0"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="ogm-jpa-tutorial" transaction-type="JTA">
        <!-- Use Hibernate OGM provider: configuration will be transparent -->
        <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
        <properties>
               <!-- property is optional if you want to use Infinispan, otherwise adjust to
 your favorite
                NoSQL Datastore provider.
            <property name="hibernate.ogm.datastore.provider" value="infinispan"/>
            -->
            <!-- defines which JTA Transaction we plan to use -->
            <property name="hibernate.transaction.jta.platform"
               value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform"/
>
        </properties>
    </persistence-unit>
```

```
</persistence>
```

Let’s now persist a set of entities and retrieve them.

```
//accessing JBoss's Transaction can be done differently but this one works nicely
TransactionManager tm = getTransactionManager();

//build the EntityManagerFactory as you would build in in Hibernate ORM
EntityManagerFactory emf = Persistence.createEntityManagerFactory(
    "ogm-jpa-tutorial");

final Logger logger = LoggerFactory.getLogger(DogBreedRunner.class);

[..]

//Persist entities the way you are used to in plain JPA
tm.begin();
logger.infof("About to store dog and breed");
EntityManager em = emf.createEntityManager();
Breed collie = new Breed();
collie.setName("Collie");
em.persist(collie);
Dog dina = new Dog();
dina.setName("Dina");
dina.setBreed(collie);
em.persist(dina);
Long dinaId = dina.getId();
em.flush();
em.close();
tm.commit();

[..]

//Retrieve your entities the way you are used to in plain JPA
tm.begin();
logger.infof("About to retrieve dog and breed");
em = emf.createEntityManager();
dina = em.find(Dog.class, dinaId);
logger.infof("Found dog %s of breed %s", dina.getName(), dina.getBreed().getName());
em.flush();
em.close();
tm.commit();

[..]

emf.close();

private static final String JBOSS_TM_CLASS_NAME = "com.arjuna.ats.jta.TransactionManager";

public static TransactionManager getTransactionManager() throws Exception {
    Class<?> tmClass = Main.class.getClassLoader().loadClass(JBOSS_TM_CLASS_NAME);
    return (TransactionManager) tmClass.getMethod("transactionManager").invoke(null);
}
```

Note

Some JVM do not handle mixed IPv4/IPv6 stacks properly (older *Mac OS X JDK in particular* [http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7144274]), if you experience trouble starting the Infinispan cluster, pass the following property: `-Djava.net.preferIPv4Stack=true` to your JVM or upgrade to a recent JDK version. jdk7u6 (b22) is known to work on Max OS X.

Note

There are some additional constraints related to transactions when working with Neo4j. You will find more details in the Neo4j transactions section: *Section 12.4, "Transactions"*

A working example can be found in Hibernate OGM's distribution under `hibernate-ogm-documentation/examples/gettingstarted`.

What have we seen?

- Hibernate OGM is a JPA implementation and is used as such both for mapping and in API usage
- It is configured as a specific JPA provider: `org.hibernate.ogm.jpa.HibernateOgmPersistence`

Let's explore more in the next chapters.

Architecture

Note

Hibernate OGM defines an abstraction layer represented by `DatastoreProvider` and `GridDialect` to separate the OGM engine from the datastores interaction. It has successfully abstracted various key/value stores, document stores and graph databases. We are working on testing it on other NoSQL families.

In this chapter we will explore:

- the general architecture
- how the data is persisted in the NoSQL datastore
- how we support JP-QL queries

Let’s start with the general architecture.

3.1. General architecture

Hibernate OGM is really made possible by the reuse of a few key components:

- Hibernate ORM for JPA support
- the NoSQL drivers to interact with the underlying datastore
- optionally Hibernate Search for indexing and query purposes
- optionally Infinispan’s Lucene Directory to store indexes in Infinispan itself, or in many other NoSQL using Infinispan’s write-through cachestores
- Hibernate OGM itself

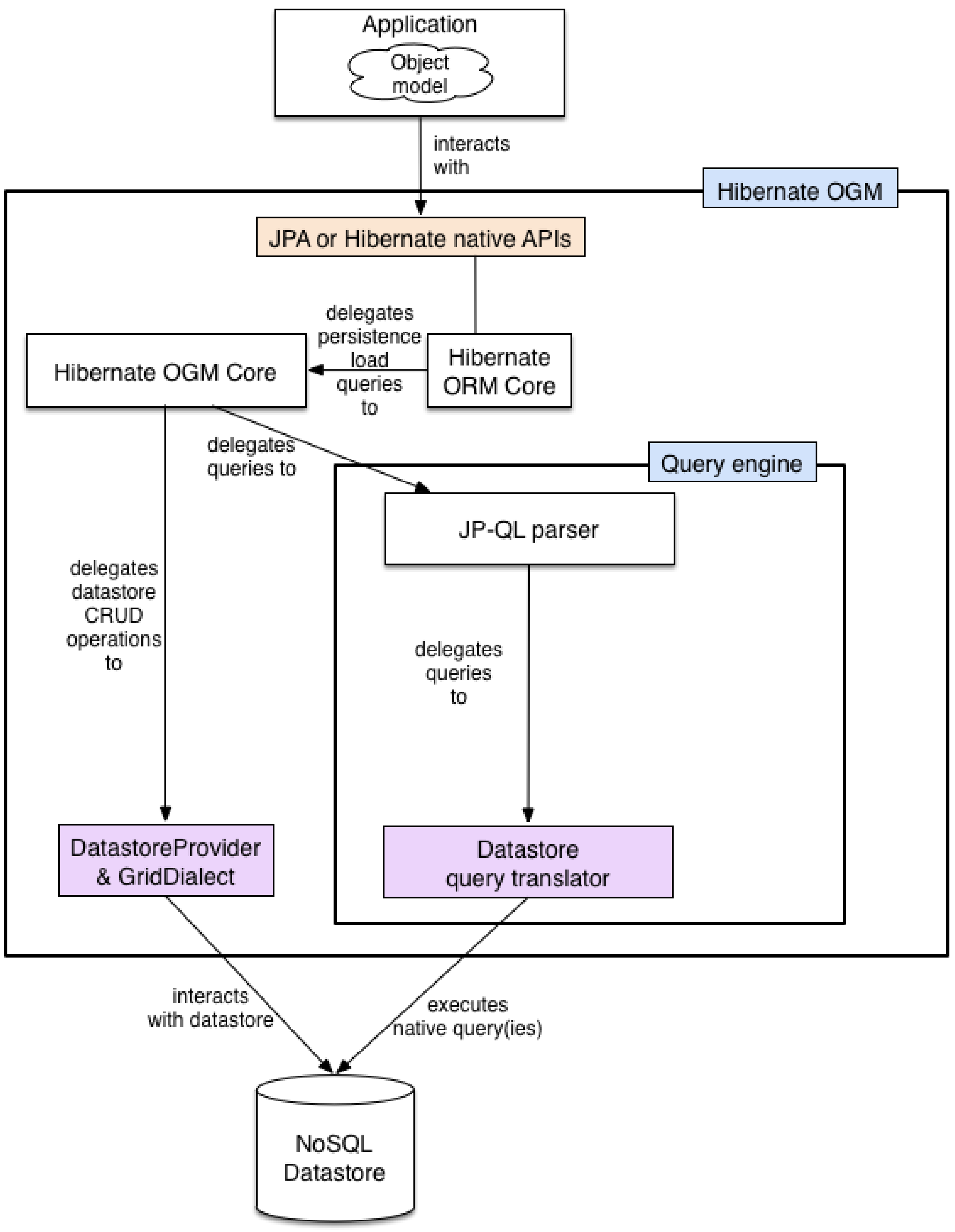


Figure 3.1. General architecture

Hibernate OGM reuses as much as possible from the Hibernate ORM infrastructure. There is no need to rewrite an entirely new JPA engine. The `Persisters` and the `Loaders` (two interfaces used by Hibernate ORM) have been rewritten to persist data in the NoSQL store. These implementations are the core of Hibernate OGM. We will see in *Section 3.2, “How is data persisted”* how the data is structured.

The particularities between NoSQL stores are abstracted by the notion of a `DatastoreProvider` and a `GridDialect`.

- `DatastoreProvider` abstracts how to start and maintain a connection between Hibernate OGM and the datastore.
- `GridDialect` abstracts how data itself including association is persisted.

Think of them as the JDBC layer for our NoSQL stores.

Other than these, all the Create/Read/Update/Delete (CRUD) operations are implemented by the Hibernate ORM engine (object hydration and dehydration, cascading, lifecycle etc).

As of today, we have implemented the following datastore providers:

- a Map based datastore provider (for testing)
- an Infinispan based datastore provider to persist your entities in Infinispan
- a Ehcache based datastore provider to persist your entities in Ehcache
- a MongoDB based datastore provider to persist data in a MongoDB database
- a Neo4j based datastore provider to persist data in the Neo4j graph database
- a CouchDB based datastore provider to persist data in the CouchDB document store

To implement JP-QL queries, Hibernate OGM parses the JP-QL string and calls the appropriate translator functions to build a native query. If the underlying engine does not have any query support, we use Hibernate Search as an external query engine.

We will discuss the subject of querying in more details in *Section 3.3, “How is data queried”*.

Hibernate OGM best works in a JTA environment. The easiest solution is to deploy it on a Java EE container. Alternatively, you can use a standalone JTA `TransactionManager`. We explain how to in *Section 4.2.2, “In a standalone JTA environment”*.

Let’s now see how and in which structure data is persisted in the NoSQL data store.

3.2. How is data persisted

Hibernate OGM tries to reuse as much as possible the relational model concepts, at least when they are practical and make sense in OGM’s case. For very good reasons, the relational model brought peace in the database landscape over 30 years ago. In particular, Hibernate OGM inherits the following traits:

- abstraction between the application object model and the persistent data model
- persist data as basic types
- keep the notion of primary key to address an entity

- keep the notion of foreign key to link two entities (not enforced)

If the application data model is too tightly coupled with your persistent data model, a few issues arise:

- any change in the application object hierarchy / composition must be reflected in the persistent data
- any change in the application object model will require a migration at the data level
- any access to the data by another application ties both applications losing flexibility
- any access to the data from another platform become somewhat more challenging
- serializing entities leads to many additional problems (see note below)

Why aren’t entities serialized in the key/value entry

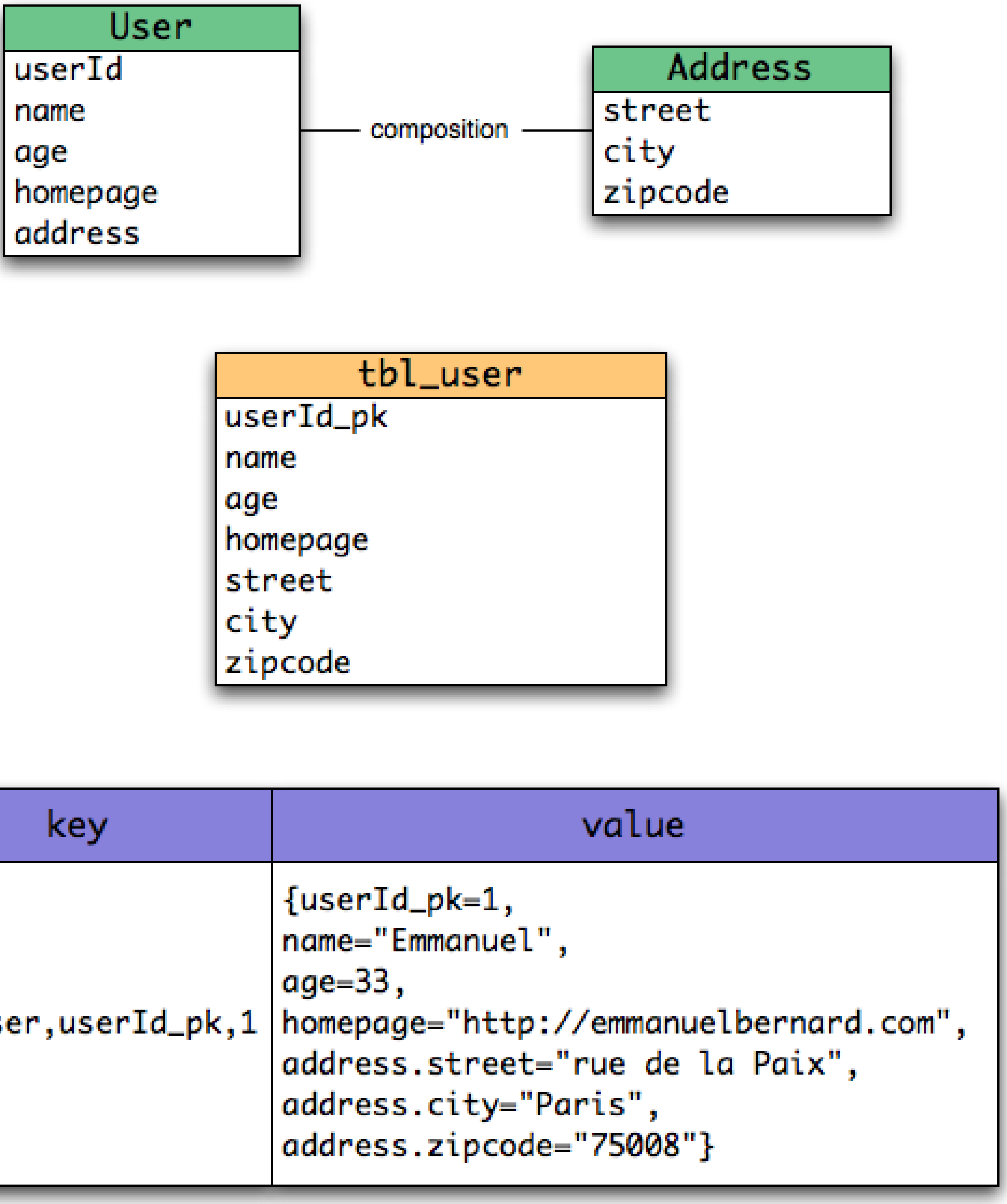
There are a couple of reasons why serializing the entity directly in the datastore - key/value in particular - can lead to problems:

- When entities are pointing to other entities are you storing the whole graph? Hint: this can be quite big!
- If doing so, how do you guarantee object identity or even consistency amongst duplicated objects? It might make sense to store the same object graph from different root objects.
- What happens in case of class schema change? If you add or remove a property or include a superclass, you must migrate all entities in your datastore to avoid deserialization issues.

Entities are stored as tuples of values by Hibernate OGM. More specifically, each entity is conceptually represented by a `Map<String,Object>` where the key represents the column name (often the property name but not always) and the value represents the column value as a basic type. We favor basic types over complex ones to increase portability (across platforms and across type / class schema evolution over time). For example a URL object is stored as its String representation.

The key identifying a given entity instance is composed of:

- the table name
- the primary key column name(s)
- the primary key column value(s)



key	value

tbl_user,userId_pk,1	{userId_pk=1,
	name="Emmanuel",
	age=33,
	homepage="http://emmanuelbernard.com",
	address.street="rue de la Paix",
	address.city="Paris",
	address.zipcode="75008"}

Figure 3.2. Storing entities

The `GridDialect` specific to the NoSQL datastore you target is then responsible to convert this map into the most natural model:

- for a key/value store or a data grid, we use the logical key as the key in the grid and we store the map as the value. Note that it's an approximation and some key/value providers will use more tailored approaches.

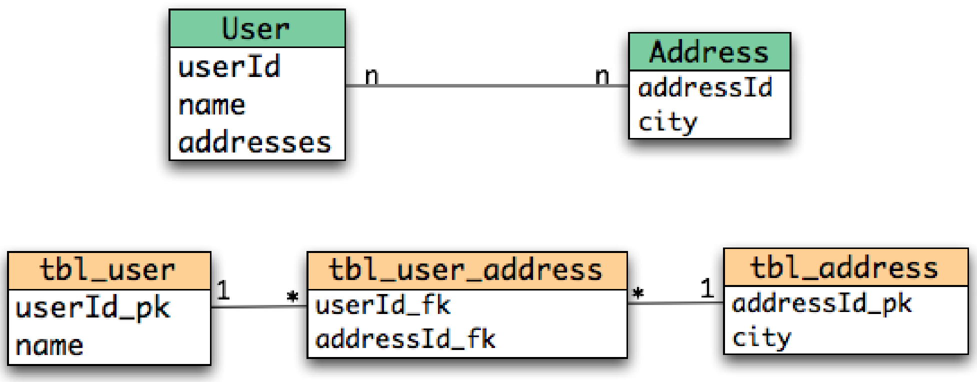
- for a document oriented store, the map is represented by a document and each entry in the map corresponds to a property in a document.

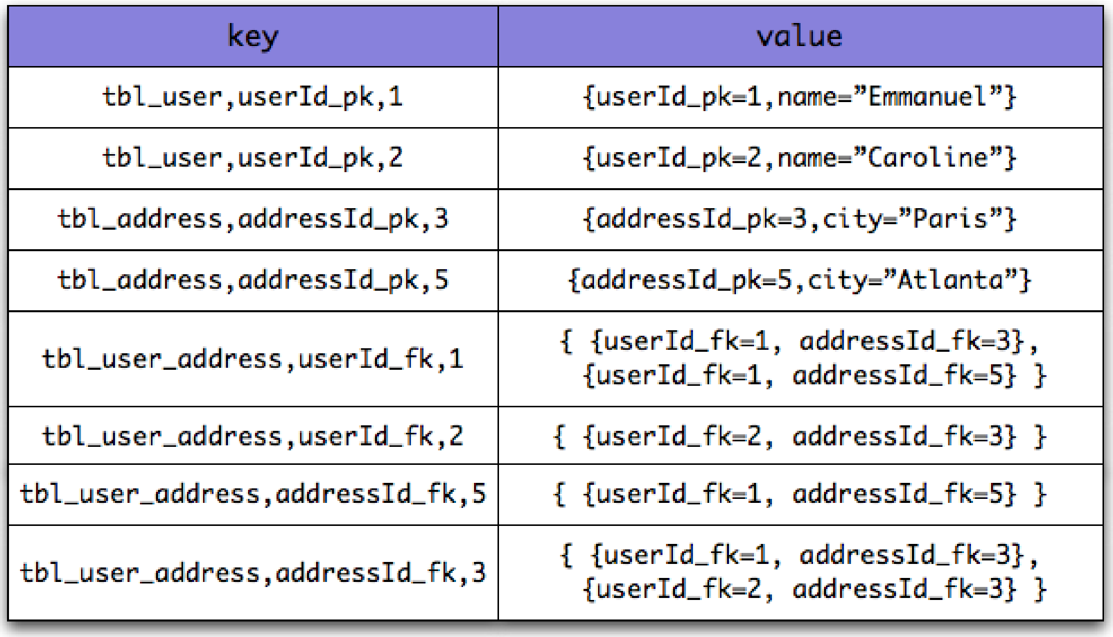
Associations are also stored as tuple as well or more specifically as a set of tuples. Hibernate OGM stores the information necessary to navigate from an entity to its associations. This is a departure from the pure relational model but it ensures that association data is reachable via key lookups based on the information contained in the entity tuple we want to navigate from. Note that this leads to some level of duplication as information has to be stored for both sides of the association.

The key in which association data are stored is composed of:

- the table name
- the column name(s) representing the foreign key to the entity we come from
- the column value(s) representing the foreign key to the entity we come from

Using this approach, we favor fast read and (slightly) slower writes.





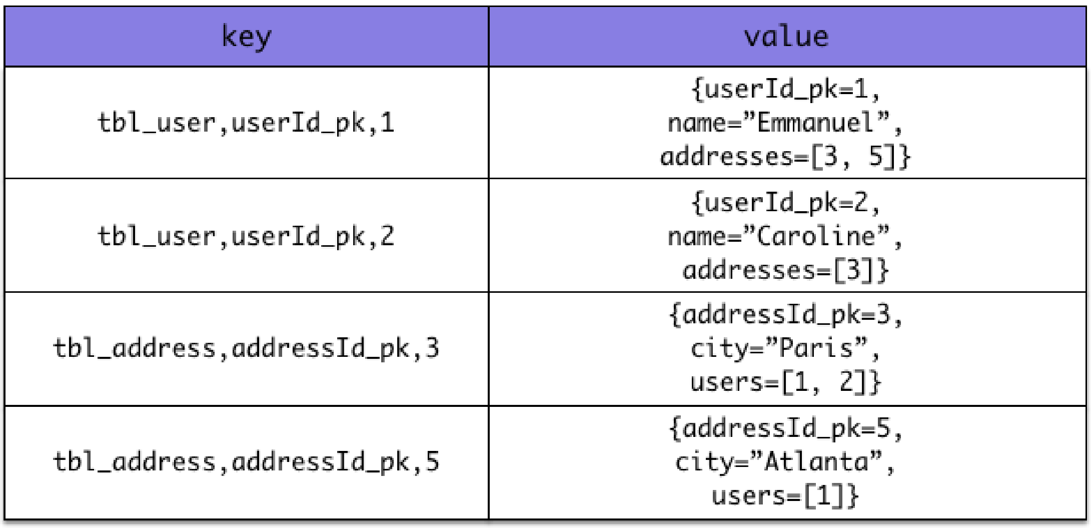
key	value
tbl_user,userId_pk,1	{userId_pk=1,name="Emmanuel"}
tbl_user,userId_pk,2	{userId_pk=2,name="Caroline"}
tbl_address,addressId_pk,3	{addressId_pk=3,city="Paris"}
tbl_address,addressId_pk,5	{addressId_pk=5,city="Atlanta"}
tbl_user_address,userId_fk,1	{ {userId_fk=1, addressId_fk=3}, {userId_fk=1, addressId_fk=5} }
tbl_user_address,userId_fk,2	{ {userId_fk=2, addressId_fk=3} }
tbl_user_address,addressId_fk,5	{ {userId_fk=1, addressId_fk=5} }
tbl_user_address,addressId_fk,3	{ {userId_fk=1, addressId_fk=3}, {userId_fk=2, addressId_fk=3} }

Figure 3.3. Storing associations

Note that this approach has benefits and drawbacks:

- it ensures that all CRUD operations are doable via key lookups
- it favors reads over writes (for associations)
- but it duplicates data

Again, there are specificities in how data is inherently stored in the specific NoSQL store. For example, in document oriented stores, the association information including the identifier to the associated entities can be stored in the entity owning the association. This is a more natural model for documents.



key	value
tbl_user,userId_pk,1	{userId_pk=1, name="Emmanuel", addresses=[3, 5]}
tbl_user,userId_pk,2	{userId_pk=2, name="Caroline", addresses=[3]}
tbl_address,addressId_pk,3	{addressId_pk=3, city="Paris", users=[1, 2]}
tbl_address,addressId_pk,5	{addressId_pk=5, city="Atlanta", users=[1]}

Figure 3.4. Storing associations in a document store

Some identifiers require to store a seed in the datastore (like sequences for examples). The seed is stored in the value whose key is composed of:

- the table name
- the column name representing the segment
- the column value representing the segment

Warning

This description is how conceptually Hibernate OGM asks the datastore provider to store data. Depending on the family and even the specific datastore, the storage is optimized to be as natural as possible. In other words as you would have stored the specific structure naturally. Make sure to check the chapter dedicated to the NoSQL store you target to find the specificities.

Many NoSQL stores have no notion of schema. Likewise, the tuple stored by Hibernate OGM is not tied to a particular schema: the tuple is represented by a `Map`, not a typed `Map` specific to a given entity type. Nevertheless, JPA does describe a schema thanks to:

- the class schema
- the JPA physical annotations like `@Table` and `@Column`.

While tied to the application, it offers some robustness and explicit understanding when the schema is changed as the schema is right in front of the developers' eyes. This is an intermediary model between the strictly typed relational model and the totally schema-less approach pushed by some NoSQL families.

3.3. How is data queried

Since Hibernate OGM wants to offer all of JPA, it needs to support JP-QL queries. Hibernate OGM parses the JP-QL query string and extracts its meaning. From there, several options are available depending of the capabilities of the NoSQL store you target:

- it directly delegates the native query generation to the datastore specific query translator implementation
- it uses Hibernate Search as a query engine to execute the query

If the NoSQL datastore has some query capabilities and if the JP-QL query is simple enough to be executed by the datastore, then the JP-QL parser directly pushes the query generation to the NoSQL specific query translator. The query returns the list of matching entity columns or projections and Hibernate OGM returns managed entities.

Some NoSQL stores have poor query support, or none at all. In this case Hibernate OGM can use Hibernate Search as its indexing and query engine. Hibernate Search is able to index and query objects - entities - and run full-text queries. It uses the well known Apache Lucene to do that but adds a few interesting characteristics like clustering support and an object oriented abstraction including an object oriented query DSL. Let’s have a look at the architecture of Hibernate OGM when using Hibernate Search:

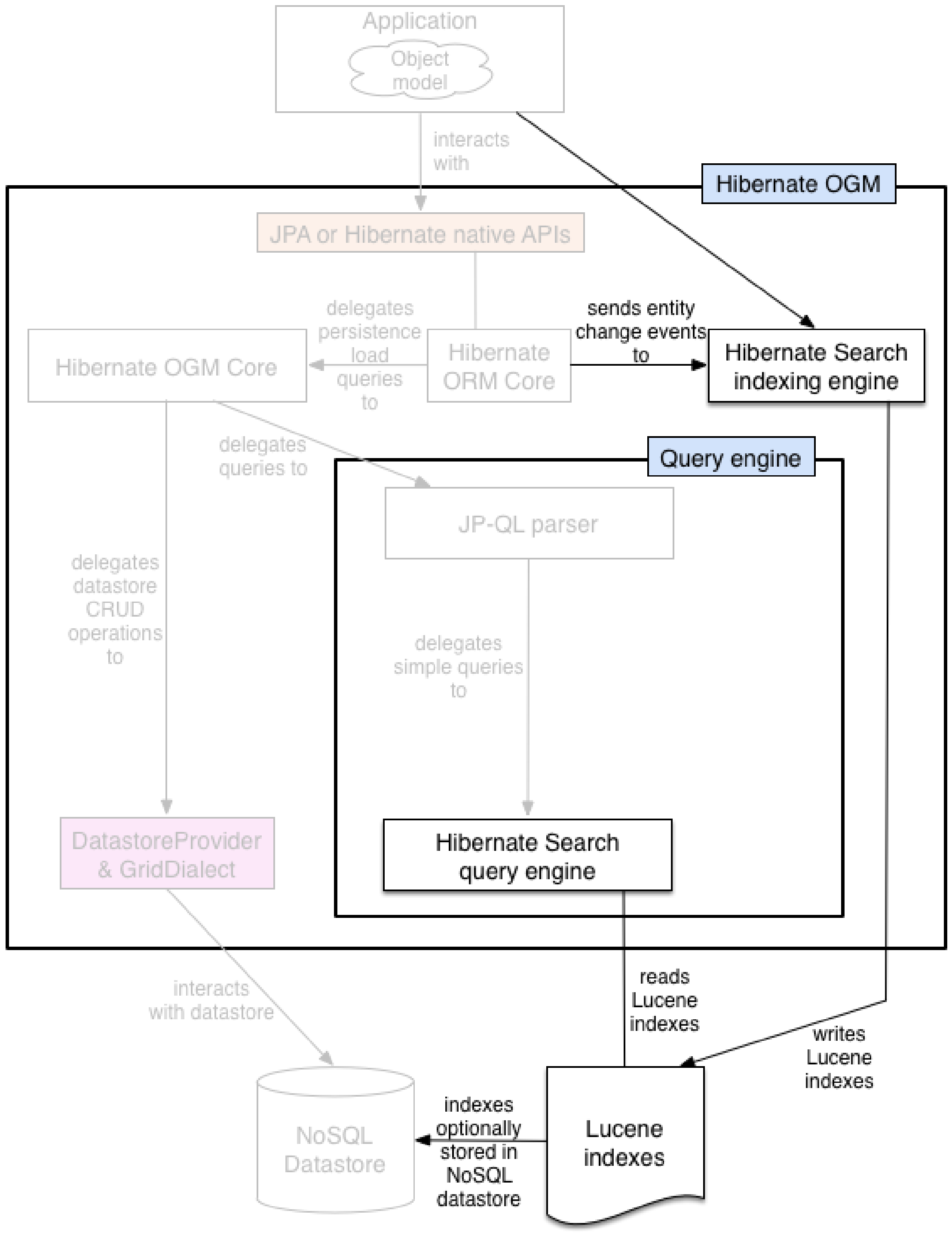

Application
Object model
interacts with
Hibernate OGM
JPA or Hibernate native APIs
delegates persistence load queries to
sends entity change events to
Hibernate OGM Core
Hibernate ORM Core
Hibernate Search indexing engine
delegates queries to
Query engine
JP-QL parser
delegates datastore CRUD operations to
delegates simple queries to
DatastoreProvider & GridDialect
Hibernate Search query engine
interacts with datastore
reads Lucene indexes
writes Lucene indexes
NoSQL Datastore
indexes optionally stored in NoSQL datastore
Lucene indexes

Figure 3.5. Using Hibernate Search as query engine - greyed areas are blocks already present in Hibernate OGM's architecture

In this situation, Hibernate ORM Core pushes change events to Hibernate Search which will index entities accordingly and keep the index and the datastore in sync. The JP-QL query parser delegates the query translation to the Hibernate Search query translator and executes the query on top of the Lucene indexes. Indexes can be stored in various fashions:

- on a file system (the default in Lucene)
- in Infinispan via the Infinispan Lucene directory implementation: the index is then distributed across several servers transparently
- in NoSQL stores like Voldemort that can natively store Lucene indexes
- in NoSQL stores that can be used as overflow to Infinispan: in this case Infinispan is used as an intermediary layer to serve the index efficiently but persists the index in another NoSQL store.

Tip

You can use Hibernate Search even if you do plan to use the NoSQL datastore query capabilities. Hibernate Search offers a few interesting options:

- clusterability
- full-text queries - ie Google for your entities
- geospatial queries
- query faceting (ie dynamic categorization of the query results by price, brand etc)

Configure and start Hibernate OGM

Hibernate OGM favors ease of use and convention over configuration. This makes its configuration quite simple by default.

4.1. Bootstrapping Hibernate OGM

Hibernate OGM can be used via the Hibernate native APIs (`Session`) or via the JPA APIs (`EntityManager`). Depending of your choice, the bootstrapping strategy is slightly different.

4.1.1. Using JPA

The good news is that if you use JPA as your primary API, the configuration is extremely simple. Hibernate OGM is seen as a persistence provider which you need to configure in your `persistence.xml`. That's it! The provider name is `org.hibernate.ogm.jpa.HibernateOgmPersistence`.

Example 4.1. persistence.xml file

```
<?xml version="1.0"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.ogm.tutorial.jpa" transaction-type="JTA">
        <!-- Use Hibernate OGM provider: configuration will be transparent -->
        <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
        <properties>
            <property name="hibernate.transaction.jta.platform"
               value="org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform" /
>
            <property name="hibernate.ogm.datastore.provider"
                      value="infinispan" />
        </properties>
    </persistence-unit>
</persistence>
```

There are a couple of things to notice:

- there is no JDBC dialect setting
- there is no JDBC setting except sometimes a `jta-data-source` (check *Section 4.2.1, “In a Java EE container”* for more info)
- there is no DDL scheme generation options (`hbm2ddl`) as NoSQL generally do not require schemas

- if you use JTA (which we recommend), you will need to set the JTA platform

You also need to configure which NoSQL datastore you want to use and how to connect to it. We will detail how to do that later in *Chapter 8, NoSQL datastores*. In this case, we have used the defaults settings for Infinispan.

From there, simply bootstrap JPA the way you are used to with Hibernate ORM:

- via `Persistence.createEntityManagerFactory`
- by injecting the `EntityManager` / `EntityManagerFactory` in a Java EE container
- by using your favorite injection framework (CDI - Weld, Spring, Guice)

4.1.2. Using Hibernate ORM native APIs

If you want to bootstrap Hibernate OGM using the native Hibernate APIs, use the class `org.hibernate.ogm.cfg.OgmConfiguration`.

Example 4.2. Bootstrap Hibernate OGM with Hibernate ORM native APIs

```
Configuration cfg = new OgmConfiguration();

//assuming you are using JTA in a non contained environment
cfg.setProperty(environment.TRANSACTION_STRATEGY,
                 "org.hibernate.transaction.JTATransactionFactory");
//assuming JBoss TransactionManager in standalone mode
cfg.setProperty(Environment.JTA_PLATFORM,
     "org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform");

//assuming the default infinispan settings
cfg.setProperty("hibernate.ogm.datastore.provider",
                "infinispan");

//add your annotated classes
cfg.addAnnotatedClass(Order.class)
   .addAnnotatedClass(Item.class)

//build the SessionFactory
SessionFactory sf = cfg.buildSessionFactory();
```

There are a couple of things to notice:

- there is no DDL schema generation options (`hbm2ddl`) as Infinispan does not require schemas
- you need to set the right transaction strategy and the right transaction manager lookup strategy if you use a JTA based transaction strategy (see *Section 4.2, “Environments”*)

You also need to configure which NoSQL datastore you want to use and how to connect to it. We will detail how to do that later in *Chapter 8, NoSQL datastores*. In this case, we have used the defaults settings for Infinispan.

4.2. Environments

Hibernate OGM runs in various environments, pretty much what you are used to with Hibernate ORM. There are however environments where it works better and has been more thoroughly tested.

4.2.1. In a Java EE container

You don't have to do much in this case. You need three specific settings:

- the transaction factory
- the JTA platform
- a JTA datasource

If you use JPA, simply set the `transaction-type` to `JTA` and the transaction factory will be set for you.

If you use Hibernate ORM native APIs only, then set `hibernate.transaction.factory_class` to either:

- `org.hibernate.transaction.CMTTransactionFactory` if you use declarative transaction demarcation.
- or `org.hibernate.transaction.JTATransactionFactory` if you manually demarcate transaction boundaries

Set the JTA platform to the right Java EE container. The property is `hibernate.transaction.transaction.jta.platform` and must contain the fully qualified class name of the lookup implementation. The list of available values are listed in *Hibernate ORM's configuration section* [http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html_single/#services-JtaPlatform]. For example, in WildFly, use `org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform`.

In your `persistence.xml`, you also need to define an existing datasource. It is not needed by Hibernate OGM and won't be used but the JPA specification mandates this setting.

Example 4.3. persistence.xml file

```
<?xml version="1.0"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.ogm.tutorial.jpa" transaction-type="JTA">
        <!-- Use Hibernate OGM provider: configuration will be transparent -->
        <provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>
        <jta-data-source>java:/DefaultDS</jta-data-source>
        <properties>
            <property name="hibernate.transaction.jta.platform"
               value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
            <property name="hibernate.ogm.datastore.provider"
                      value="infinispan" />
        </properties>
    </persistence-unit>
</persistence>
```

`java:DefaultDS` will work for out of the box WildFly deployments.

4.2.2. In a standalone JTA environment

There is a set of common misconceptions in the Java community about JTA:

- JTA is hard to use
- JTA is only needed when you need transactions spanning several databases
- JTA works in Java EE only
- JTA is slower than "simple" transactions

None of that is true of course, let me show you how to use JBoss Transaction in a standalone environment with Hibernate OGM.

In Hibernate OGM, make sure to set the following properties:

- `transaction-type` to `JTA` in your persistence.xml if you use JPA
- or `hibernate.transaction.factory_class` to `org.hibernate.transaction.JTATransactionFactory` if you use `OgmConfiguration` to bootstrap Hibernate OGM.
- `hibernate.transaction.jta.platform` to `org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform` in both cases.

On the JBoss Transaction side, add JBoss Transaction in your classpath. If you use maven, it should look like this:

Example 4.4. JBoss Transaction dependency declaration

```
<dependency>
    <groupId>org.jboss.jbossts</groupId>
    <artifactId>jbossjta</artifactId>
    <version>4.16.4.Final</version>
</dependency>
```

The next step is you get access to the transaction manager. The easiest solution is to do as the following example:

```
TransactionManager transactionManager =
   com.arjuna.ats.jta.TransactionManager.transactionmanager();
```

Then use the standard JTA APIs to demarcate your transaction and you are done!

Example 4.5. Demarcate your transaction with standalone JTA

```
//note that you must start the transaction before creating the EntityManager
//or else call entityManager.joinTransaction()
transactionManager.begin();

final EntityManager em = emf.createEntityManager();

Poem poem = new Poem();
poem.setName("L'albatros");
em.persist(poem);

transactionManager.commit();

em.clear();

transactionManager.begin();

poem = em.find(Poem.class, poem.getId());
assertThat(poem).isNotNull();
assertThat(poem.getName()).isEqualTo("L'albatros");
em.remove(poem );

transactionManager.commit();

em.close();
```

That was not too hard, was it? Note that application frameworks like Seam or Spring Framework should be able to initialize the transaction manager and call it to demarcate transactions for you. Check their respective documentation.

4.2.3. Without JTA

While this approach works today, it does not ensure that works are done transactionally and hence won't be able to rollback your work. This will change in the future but in the mean time, such an environment is not recommended.

Note

For NoSQL datastores not supporting transactions, this is less of a concern.

4.3. Configuration options

The most important options when configuring Hibernate OGM are related to the datastore. They are explained in *Chapter 8, NoSQL datastores*.

Otherwise, most options from Hibernate ORM and Hibernate Search are applicable when using Hibernate OGM. You can pass them as you are used to do either in your `persistence.xml` file, your `hibernate.cfg.xml` file or programmatically.

More interesting is a list of options that do *not* apply to Hibernate OGM and that should not be set:

- `hibernate.dialect`
- `hibernate.connection.*` and in particular `hibernate.connection.provider_class`
- `hibernate.show_sql` and `hibernate.format_sql`
- `hibernate.default_schema` and `hibernate.default_catalog`
- `hibernate.use_sql_comments`
- `hibernate.jdbc.*`
- `hibernate.hbm2ddl.auto` and `hibernate.hbm2ddl.import_file`

4.4. Configuring Hibernate Search

Hibernate Search integrates with Hibernate OGM just like it does with Hibernate ORM. The Hibernate Search version tested is 5.1.0.Final. Add the dependency to your project - the group id is `org.hibernate` and artifact id `hibernate-search-orm`.

Then configure where you want to store your indexes, map your entities with the relevant index annotations and you are good to go. For more information, simply check the *Hibernate Search reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/].

In *Section 9.6, "Storing a Lucene index in Infinispan"* we'll discuss how to store your Lucene indexes in Infinispan. This is useful even if you don't plan to use Infinispan as your primary data store.

4.5. How to package Hibernate OGM applications for WildFly 8.2

Provided you're deploying on WildFly 8.2, there is an additional way to add the OGM dependencies to your application.

In WildFly 8.2, class loading is based on modules that have to define explicit dependencies on other modules. Modules allow to share the same artifacts across multiple applications, getting you smaller and quicker deployments.

More details about modules are described in *Class Loading in WildFly* [https://docs.jboss.org/author/display/WFLY8/Class+Loading+in+WildFly].

4.5.1. Packaging Hibernate OGM applications for WildFly 8.2

You can download the pre-packaged module ZIP from:

- *Sourceforge* [https://downloads.sourceforge.net/project/hibernate/hibernate-ogm/4.1.3.Final/hibernate-ogm-modules-wildfly8-4.1.3.Final.zip]
- *JBoss's Maven repository* [https://repository.jboss.org/nexus/service/local/artifact/maven/redirect?r=central&g=org.hibernate.ogm&a=hibernate-ogm-modules-wildfly8&v=4.1.3.Final&e=zip]

Unpack the archive into the `modules` folder of your WildFly 8.2 installation. The modules included are:

- *org.hibernate:ogm*, the core Hibernate OGM library and the Infinispan datastore provider.
- *org.hibernate.ogm.<%DATASTORE%>:main*, one module for each datastore provider besides Infinispan, with *<%DATASTORE%>* being one of *ehcache*, *mongodb* etc. You only need to add those modules which you actually intend to use.
- Several shared dependencies such as *org.hibernate.hql:<%VERSION%>* (containing the query parser) and others

There are two ways to include the dependencies in your project:

Using the manifest
: Add this entry to the MANIFEST.MF in your archive (replace *<%DATASTORE%>* with the right value for your chosen datastore):

```
Dependencies: org.hibernate:ogm services, org.hibernate.ogm.<%DATASTORE
%>:main services
```

Using jboss-deployment-structure.xml
: This is a JBoss-specific descriptor. Add a `WEB-INF/jboss-deployment-structure.xml` in your archive with the following content (replace *<%DATASTORE%>* with the right value for your chosen datastore):

```
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.hibernate" slot="ogm" services="export" />
            <module name="org.hibernate.ogm.<%DATASTORE%>" slot="main" services="export" />
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

More information about the descriptor can be found in the *WildFly documentation* [https://docs.jboss.org/author/display/WFLY8/Class+Loading+in+WildFly].

4.5.2. Loading both the Hibernate Search and Hibernate OGM modules WildFly 8.2

The Hibernate OGM module does not include the Hibernate Search module, so this will need to be downloaded separately.

The Hibernate Search documentation has a similar section describing the details: *Update and activate latest Hibernate Search version in WildFly* [http://docs.jboss.org/hibernate/search/5.1/reference/en-US/html/search-configuration.html#_update_and_activate_latest_hibernate_search_version_in_wildfly].

If your application needs to use both Hibernate OGM and Hibernate Search, your MANIFEST.MF will look like:

Example 4.6. Example MANIFEST.MF for an application using Hibernate OGM for CouchDB and also Hibernate Search

```
org.hibernate:ogm services, org.hibernate.ogm.couchdb services,
 org.hibernate.search.orm:5.1 services
```

Map your entities

This section mainly describes the specificities of Hibernate OGM mappings. It is not be a comprehensive guide to entity mappings, the complete guide is *Hibernate ORM's documentation* [http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch05.html]: after all Hibernate OGM *is* Hibernate ORM.

5.1. Supported entity mapping

Pretty much all entity related constructs should work out of the box in Hibernate OGM. `@Entity`, `@Table`, `@Column`, `@Enumarated`, `@Temporal`, `@Cacheable` and the like will work as expected. If you want an example, check out *Chapter 2, Getting started with Hibernate OGM* or the documentation of Hibernate ORM. Let's concentrate of the features that differ or are simply not supported by Hibernate OGM.

The various inheritance strategies are not supported by Hibernate OGM, only the table per concrete class strategy is used. This is not so much a limitation but rather an acknowledgment of the dynamic nature of NoSQL schemas. If you feel the need to support other strategies, let us know (see *Section 1.2, "How to contribute"*). Simply do not use `@Inheritance` nor `@DiscriminatorColumn`.

Secondary tables are not supported by Hibernate OGM at the moment. If you have needs for this feature, let us know (see *Section 1.2, "How to contribute"*).

Queries are partially supported, you will find more information in the *query chapter*.

All standard JPA id generators are supported: IDENTITY, SEQUENCE, TABLE and AUTO. If you need support for additional generators, let us know (see *Section 1.2, "How to contribute"*).

Note

We recommend you use a UUID based generator as this type of generator allows maximum scalability to the underlying datastore as no cluster-wide counter is necessary.

```
@Entity
public class Breed {

    @Id @GeneratedValue(generator = "uuid")
    @GenericGenerator(name="uuid", strategy="uuid2")
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    private String id;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    private String name;
}
```

5.2. Supported Types

Most Java built-in types as supported at this stage. However, custom types (`@Type`) are not supported.

Here is a list of supported Java types:

- Boolean
- Byte
- Byte Array
- Calendar
- Class
- Date
- Double
- Integer
- Long
- Short
- Float
- Character
- String
- BigDecimal (mapped as scientific notation)
- BigInteger
- Url (as described by RFC 1738 and returned by toString of the Java URL type)
- UUID stored as described by RFC 4122
- Enums

Let us know if you need more type support *Section 1.2, “How to contribute”*

5.3. Supported association mapping

All association types are supported (`@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`). Likewise, all collection types are supported (`Set`, `Map`, `List`). The way Hibernate OGM stores association information is however quite different than the traditional RDBMS representation. Each chapter dedicated to a datastore describes how associations are persisted, make sure to check them out.

Keep in mind that collections with many entries won’t perform very well in Hibernate OGM (at least today) as all of the association navigation for a given entity is stored in a single key. If your collection is made of 1 million elements, Hibernate OGM stores 1 million tuples in the association key.

Hibernate OGM APIs

Hibernate OGM has very few specific APIs. For the most part, you will interact with it via either:

- the JPA APIs
- the native Hibernate ORM APIs

This chapter will only discuss the Hibernate OGM specific behaviors regarding these APIs. If you need to learn JPA or the native Hibernate APIs, check out *the Hibernate ORM documentation* [http://hibernate.org/orm/documentation/].

6.1. Bootstrap Hibernate OGM

We already discussed this subject earlier, have a look at *Chapter 4, Configure and start Hibernate OGM* for all the details.

As a reminder, it basically boils down to either:

- set the right persistence provider in your `persistence.xml` file and create an `EntityManagerFactory` the usual way
- start via the Hibernate ORM native APIs using `OgmConfiguration` to boot a `SessionFactory`

6.2. JPA and native Hibernate ORM APIs

You know of the Java Persistence and Hibernate ORM native APIs? You are pretty much good to go. If you need a refresher, make sure you read the *Hibernate ORM documentation* [http://hibernate.org/orm/documentation/].

A few things are a bit different though, let's discuss them.

Most of the `EntityManager` and `Session` contracts are supported. Here are the few exceptions:

- `Session.createCriteria`: criteria queries are not yet supported in Hibernate OGM
- `Session.createFilter`: queries on collections are not supported yet
- `Session` 's `enableFilter`, `disableFilter` etc: query filters are not supported at the moment
- `doWork` and `doReturningWork` are not implemented as they rely on JDBC connections - see *OGM-694* [https://hibernate.atlassian.net/browse/OGM-694]
- `Session` 's stored procedure APIs are not supported
- `Session` 's natural id APIs are not yet supported
- `Session.lock` is not fully supported at this time
- `EntityManager` 's criteria query APIs are not supported

- `EntityManager` 's stored procedure APIs are not supported - see *OGM-695* [https://hibernate.atlassian.net/browse/OGM-695]
- `EntityManager.lock` is not fully supported at this time
- see *Chapter 7, Query your entities* to know what is supported for JP-QL and native queries

6.2.1. Accessing the `OgmSession` API

To execute NoSQL native queries, one approach is to use `OgmSession.createNativeQuery`. You can read more about it in *Section 7.2, "Using the native query language of your NoSQL"*. But let's see how to access an `OgmSession` instance.

From JPA, use the `unwrap` method of `EntityManager`

Example 6.1. Get to an `OgmSession` from an `EntityManager`

```
EntityManager entityManager = ...
OgmSession ogmSession = entityManager.unwrap(OgmSession.class);
NoSQLQuery query = ogmSession.createNativeQuery(...);
```

In the Hibernate native API case, you should already have access to an `OgmSession`. The `OgmConfiguration` you used returns an `OgmSessionFactory`. This factory in turns produces `OgmSession`.

Example 6.2. Get to an `OgmSession` with Hibernate ORM native APIs

```
OgmConfiguration ogmConfiguration = new OgmConfiguration();
OgmSessionFactory ogmSessionFactory = ogmConfiguration.buildSessionFactory();
OgmSession ogmSession = ogmSessionFactory.openSession();
NoSQLQuery query = ogmSession.createNativeQuery(...);
```

6.3. On flush and transactions

While most underlying datastores do not support transaction, it is important to demarcate transaction via the Hibernate OGM APIs. Let's see why.

Hibernate does pile up changes for as long as it can before pushing them down to the datastore. This opens up the doors to huge optimizations (avoiding duplication, batching operations etc). You can force changes to be sent to the datastore by calling `Session.flush` or `EntityManager.flush`. In some situations - for example before some queries are executed -, Hibernate will flush automatically. It will also flush when the transaction demarcation happens (whether there is a real transaction or not).

The best approach is to always demarcate the transaction as shown below. This avoids the needs to manually call flush and will offer future opportunities for Hibernate OGM.

Example 6.3. Explicitly demarcating transactions

Here is how you do outside of a JTA environment.

```
Session session = ...

Transaction transaction = session.beginTransaction();
try {
    // do your work
    transaction.commit(); // will flush changes to the datastore
catch (Exception e) {
    transaction.rollback();
}

// or in JPA
EntityManager entityManager = ...
EntityTransaction transaction = entityManager.getTransaction();
try {
    // do your work
    transaction.commit(); // will flush changes to the datastore
}
catch (Exception e) {
    transaction.rollback();
}
```

Inside a JTA environment, either the container demarcate the transaction for you and Hibernate OGM will transparently joins that transaction and flush at commit time. Or you need to manually demarcate the transaction. In the latter case, it is best to start / stop the transaction before retrieving the `Session` or `EntityManager` as show below. The alternative is to call the `EntityManager.joinTransaction()` once the transaction has started.

```
transactionManager.begin();
Session session = sessionFactory.openSession();
// do your work
transactionManager.commit(); // will flush changes to the datastore

// or in JPA
transactionManager.begin();
EntityManager entityManager = entityManagerFactory.createEntityManager();
// do your work
transactionManager.commit(); // will flush changes to the datastore
```

6.4. SPIs

Some of the Hibernate OGM public contracts are geared towards either integrators or implementors of datastore providers. They should not be used by a regular application. These contracts are named SPIs and are in a `.spi` package.

To keep improving Hibernate OGM, we might break these SPIs between versions. If you plan on writing a datastore, come and talk to us.

Tip

Non public contracts are stored within a `.impl` package. If you see yourself using one of these classes, beware that we can break these without notice.

Query your entities

Once your data is in the datastore, it's time for some query fun! With Hibernate OGM, you have a few alternatives that should get you covered:

- Use JP-QL - only for simple queries for now
- Use the NoSQL native query mapping the result as managed entities
- Use Hibernate Search queries - primarily full-text queries

7.1. Using JP-QL

For Hibernate OGM, we developed a brand new JP-QL parser which is already able to convert simple queries into the native underlying datastore query language (e.g. MongoQL for MongoDB, CypherQL for Neo4J, etc). This parser can also generate Hibernate Search queries for datastores that do not support a query language.

Note

For datastores like Infinispan that require Hibernate Search to execute JP-QL queries, the following preconditions must be met:

- no join, aggregation, or other relational operations are implied
- the entity involved in the query must be indexed
- the properties involved in the predicates must be indexed

Here is an example:

```
@Entity @Indexed
public class Hypothesis {

    @Id
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    private String id;

    @Field(analyze=Analyze.NO)
    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    private String description;
}

Query query = session
    .createQuery("from Hypothesis h where h.description = :desc")
    .setString("desc", "tomorrow it's going to rain");
```

Note that the `description` field is marked as not analysed. This is necessary to support field equality and comparison as defined by JP-QL.

You can make use of the following JP-QL constructs:

- simple comparisons using "<", "#", "=", ">=" and ">"
- `IS NULL` and `IS NOT NULL`
- the boolean operators `AND`, `OR`, `NOT`
- `LIKE`, `IN` and `BETWEEN`
- `ORDER BY`

In particular and of notice, what is not supported is:

- refer to an embedded object property *OGM-692* [https://hibernate.atlassian.net/browse/OGM-692] in a predicate
- refer to a collection of embedded objects *OGM-693* [https://hibernate.atlassian.net/browse/OGM-693]
- cross entity joins
- JP-QL functions in particular aggregation functions like `count`
- JP-QL update and delete queries

That may sound rather limiting for your use cases so bear with us. This is a hot area we want to improve, please tell us what feature you miss *by opening a JIRA or via email*. Also read the next section, you will see other alternatives to implement your queries.

Let’s look at some of the queries you can express in JP-QL:

Example 7.1. Some JP-QL queries

```
// query returning an entity based on a simple predicate
select h from Hypothesis h where id = 16

// projection of the entity property
select id, description from Hypothesis h where id = 16

// projection of the embedded properties
```

```
select h.author.address.street from Hypothesis h where h.id = 16

// predicate comparing a property value and a literal
from Hypothesis h where h.position = '2'

// negation
from Hypothesis h where not h.id = '13'
from Hypothesis h where h.position <> 4

// conjunction
from Hypothesis h where h.position = 2 and not h.id = '13'

// named parameters
from Hypothesis h where h.description = :myParam

// range query
from Hypothesis h where h.description BETWEEN :start and :end"

// comparisons
from Hypothesis h where h.position < 3

// in
from Hypothesis h where h.position IN (2, 3, 4)

// like
from Hypothesis h where h.description LIKE '%dimensions%'

// comparison with null
from Hypothesis h where h.description IS null

// order by
from Hypothesis h where h.description IS NOT null ORDER BY id
from Helicopter h order by h.make desc, h.name
```

Note

In order to reflect changes performed in the current session, all entities affected by a given query are flushed to the datastore prior to query execution (that's the case for Hibernate ORM as well as Hibernate OGM).

For not fully transactional stores, this can cause changes to be written as a side-effect of running queries which cannot be reverted by a possible later rollback.

Depending on your specific use cases and requirements you may prefer to disable auto-flushing, e.g. by invoking `query.setFlushMode(FlushMode.MANUAL)`. Bear in mind though that query results will then not reflect changes applied within the current session.

7.2. Using the native query language of your NoSQL

Often you want the raw power of the underlying NoSQL query engine. Even if that costs you portability.

Hibernate OGM addresses that problem by letting you express native queries (e.g. in MongoQL or CypherQL) and map the result of these queries as mapped entities.

In JPA, use `EntityManager.createNativeQuery`. The first form accepts a result class if your result set maps the mapping definition of the entity. The second form accepts the name of a resultSetMapping and lets you customize how properties are mapped to columns by the query. You can also used a predefined named query which defines its result set mapping.

Let's take a look at how it is done for Neo4J:

Example 7.2. Various ways to create a native query in JPA

```
@Entity
@NamedNativeQuery(
   name = "AthanasiaPoem",
   query = "{ $and: [ { name : 'Athanasia' }, { author : 'Oscar Wilde' } ] }",
   resultClass = Poem.class )
public class Poem {

    @Id
    private Long id;

    private String name;

    private String author;

   // getters, setters ...

}

...

javax.persistence.EntityManager em = ...

// a single result query
String query1 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) RETURN n";
Poem poem = (Poem) em.createNativeQuery( query1, Poem.class ).getSingleResult();

// query with order by
String query2 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) " +
                "RETURN n ORDER BY n.name";
List<Poem> poems = em.createNativeQuery( query2, Poem.class ).getResultList();

// query with projections
String query3 = MATCH ( n:Poem ) RETURN n.name, n.author ORDER BY n.name";
List<Object[]> poemNames = (List<Object[]>)em.createNativeQuery( query3 )
                               .getResultList();

// named query
```

```
Poem poem = (Poem) em.createNamedQuery( "AthanasiaPoem" ).getSingleResult();
```

In the native Hibernate API, use `OgmSession.createNativeQuery` or `Session.getNamedQuery`. The former form lets you define the result set mapping programmatically. The latter is receiving the name of a predefined query already describing its result set mapping.

Example 7.3. Hibernate API defining a result set mapping

```
OgmSession session = ...
String query1 = "{ $and: [ { name : 'Portia' }, { author : 'Oscar Wilde' } ] }";
Poem poem = session.createNativeQuery( query1 )
                      .addEntity( "Poem", Poem.class )
                      .uniqueResult();
```

Check out each individual datastore chapter for more info on the specifics of the native query language mapping. In particular *Neo4J* and *MongoDB*.

7.3. Using Hibernate Search

Hibernate Search offers a way to index Java objects into Lucene indexes and to execute full-text queries on them. The indexes do live outside your datastore. This offers a few interesting properties in terms of feature set and scalability.

Apache Lucene is a full-text indexing and query engine with excellent query performance. Feature wise, *full-text* means you can do much more than a simple equality match.

Hibernate Search natively integrates with Hibernate ORM. And Hibernate OGM of course!

Example 7.4. Using Hibernate Search for full-text matching

```
@Entity @Indexed
public class Hypothesis {

    @Id
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    private String id;

    @Field(analyze=Analyze.YES)
    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
    private String description;
}
```

```
EntityManager entityManager = ...
//Add full-text superpowers to any EntityManager:
FullTextEntityManager ftem = Search.getFullTextEntityManager(entityManager);
```

```
//Optionally use the QueryBuilder to simplify Query definition:
QueryBuilder b = ftem.getSearchFactory()
   .buildQueryBuilder()
   .forEntity(Hypothesis.class)
   .get();

//Create a Lucene Query:
Query lq = b.keyword().onField("description").matching("tomorrow").createQuery();

//Transform the Lucene Query in a JPA Query:
FullTextQuery ftQuery = ftem.createFullTextQuery(lq, Hypothesis.class);

//List all matching Hypothesis:
List<Hypothesis> resultList = ftQuery.getResultList();
```

Assuming our database contains an `Hypothesis` instance having description "Sometimes tomorrow we release", that instance will be returned by our full-text query.

Text similarity can be very powerful as it can be configured for specific languages or domain specific terminology; it can deal with typos and synonyms, and above all it can return results by *relevance*.

Worth noting the Lucene index is a vectorial space of term occurrence statistics: so extracting tags from text, frequencies of strings and correlate this data makes it very easy to build efficient data analysis applications.

While the potential of Lucene queries is very high, it's not suited for all use cases Let's see some of the limitations of Lucene Queries as our main query engine:

- Lucene doesn't support Joins. Any `to-One` relations can be mapped fine, and the Lucene community is making progress on other forms, but restrictions on `OneToMany` or `ManyToMany` can't be implemented today.
- Since we apply changes to the index at commit time, your updates won't affect queries until you commit (we might improve on this).
- While queries are extremely fast, write operations are not as fast (but we can make it scale).

For a complete understanding of what Hibernate Search can do for you and how to use it, go check the *Hibernate Search reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/].

7.4. Using the Criteria API

At this time, we have not implemented support for the Criteria APIs (neither Hibernate native and JPA).

NoSQL datastores

Currently Hibernate OGM supports the following NoSQL datastores:

- Map: stores data in an in-memory Java map to store data. Use it only for unit tests.
- Infinispan: stores data into *Infinispan* [http://infinispan.org/] (data grid)
- Ehcache: stores data into *Ehcache* [http://ehcache.org/] (cache)
- MongoDB: stores data into *MongoDB* [http://www.mongodb.org/] (document store)
- Neo4j: stores data into *Neo4j* [http://www.neo4j.org/] (graph database)
- CouchDB: stores data into *CouchDB* [https://couchdb.apache.org/] (document store)
  - at this stage, this datastore is experimental

More are planned, if you are interested, come talk to us (see *Chapter 1, How to get help and contribute on Hibernate OGM*).

For each datastore, Hibernate OGM has specific integration code called a datastore provider. All are in a dedicated maven module, you simply need to depend on the one you use.

Hibernate OGM interacts with NoSQL datastores via two contracts:

- a `DatastoreProvider` which is responsible for starting and stopping the connection(s) with the datastore and prop up the datastore if needed
- a `GridDialect` which is responsible for converting an Hibernate OGM operation into a datastore specific operation

8.1. Using a specific NoSQL datastore

Only a few steps are necessary:

- add the datastore provider module in your classpath
- ask Hibernate OGM to use that datastore provider
- configure the URL or configuration to reach the datastore

Adding the relevant Hibernate OGM module in your classpath looks like this in Maven:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan</artifactId>
    <version>4.1.3.Final</version>
</dependency>
```

The module names are `hibernate-ogm-infinispan`, `hibernate-ogm-ehcache`, `hibernate-ogm-mongodb`, `hibernate-ogm-neo4j` and `hibernate-ogm-couchdb`. The map datastore is included in the Hibernate OGM engine module.

Next, configure which datastore provider you want to use. This is done via the `hibernate.ogm.datastore.provider` option. Possible values are:

- specific shortcuts (preferable): `map` (only to be used for unit tests), `infinispan`, `ehcache`, `mongodb`, `neo4j_embedded` or `couchdb_experimental`
- the fully qualified class name of a `DatastoreProvider` implementation

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants declared on `OgmProperties` to specify configuration properties such as `hibernate.ogm.datastore.provider`.

In this case you also can specify the provider in form of a class object of a datastore provider type or pass an instance of a datastore provider type:

```
Map<String, Object> properties = new HashMap<String, Object>();

// pass the type
properties.put( OgmProperties.DATASTORE_PROVIDER, MongoDBDatastoreProvider.class );

EntityManagerFactory emf = Persistence.createEntityManagerFactory( "my-pu", properties );
```

By default, a datastore provider chooses the best grid dialect transparently but you can manually override that setting with the `hibernate.ogm.datastore.grid_dialect` option. Use the fully qualified class name of the `GridDialect` implementation. Most users should ignore this setting entirely and live a happier life instead.

Let's now look at the specifics of each datastore provider. How to configure it further, what mapping structure is used and more.

Infinispan

Infinispan is an open source in-memory data grid focusing on high performance. As a data grid, you can deploy it on multiple servers - referred to as nodes - and connect to it as if it were a single storage engine: it will cleverly distribute both the computation effort and the data storage.

It is trivial to setup on a single node, in your local JVM, so you can easily try Hibernate OGM. But Infinispan really shines in multiple node deployments: you will need to configure some networking details but nothing changes in terms of application behaviour, while performance and data size can scale linearly.

From all its features we will only describe those relevant to Hibernate OGM; for a complete description of all its capabilities and configuration options, refer to the Infinispan project documentation at *infinispan.org* [http://infinispan.org].

9.1. Configure Infinispan

You configure Hibernate OGM and Infinispan in two steps basically:

- Add the dependencies to your classpath
- And then choose one of:
  - Use the default Infinispan configuration (no action needed)
  - Point to your own configuration resource file
  - Point to a `JNDI` name of an existing Infinispan instance

9.1.1. Adding Infinispan dependencies

To add the dependencies via Maven, add the following module:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan</artifactId>
    <version>4.1.3.Final</version>
</dependency>
```

If you’re not using a dependency management tool, copy all the dependencies from the distribution in the directories:

- `/lib/required`
- `/lib/infinispan`
- Optionally - depending on your container - you might need some of the jars from `/lib/provided`

9.1.2. Infinispan specific configuration properties

The advanced configuration details of an Infinispan Cache are defined in an Infinispan specific XML configuration file; the Hibernate OGM properties are simple and usually just point to this external resource.

To use the default configuration provided by Hibernate OGM - which is a good starting point for new users - you don't have to set any property.

Hibernate OGM properties for Infinispan

`hibernate.ogm.datastore.provider`
: Set it to `infinispan` to use Infinispan as the datastore provider.

`hibernate.ogm.infinispan.cachemanager_jndi_name`
: If you have an Infinispan `EmbeddedCacheManager` registered in JNDI, provide the JNDI name and Hibernate OGM will use this instance instead of starting a new `CacheManager`. This will ignore any further configuration properties as Infinispan is assumed being already configured. Infinispan can typically be pushed to JNDI via WildFly, Spring or Seam.

`hibernate.ogm.infinispan.configuration_resource_name`
: Should point to the resource name of an Infinispan configuration file. This is ignored in case `JNDI` lookup is set. Defaults to `org/hibernate/ogm/datastore/infinispan/default-config.xml`.

`hibernate.ogm.datastore.keyvalue.cache_storage`
: The strategy for persisting data in Infinispan. The following two strategies exist (values of the `org.hibernate.ogm.datastore.keyvalue.options.CacheMappingType` enum):

  - `CACHE_PER_TABLE`: A dedicated cache will be used for each entity type, association type and id source table.
  - `CACHE_PER_KIND`: Three caches will be used: one cache for all entities, one cache for all associations and one cache for all id sources.

  Defaults to `CACHE_PER_TABLE`. It is the recommended strategy as it makes it easier to target a specific cache for a given entity.

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `InfinispanProperties` when specifying the configuration properties listed above.

Common properties shared between stores are declared on `OgmProperties` (a super interface of `InfinispanProperties`).

For maximum portability between stores, use the most generic interface possible.

9.1.3. Cache names used by Hibernate OGM

Depending on the cache mapping approach, Hibernate OGM will either:

- store each entity type, association type and id source table in a dedicated cache very much like what Hibernate ORM would do. This is the `CACHE_PER_TABLE` approach.
- store data in three different caches when using the `CACHE_PER_KIND` approach:
  - `ENTITIES`: is going to be used to store the main attributes of all your entities.
  - `ASSOCIATIONS`: stores the association information representing the links between entities.
  - `IDENTIFIER_STORE`: contains internal metadata that Hibernate OGM needs to provide sequences and auto-incremental numbers for primary key generation.

The preferred strategy is `CACHE_PER_TABLE` as it offers both more fine grained configuration options and the ability to work on specific entities in a more simple fashion.

In the following paragraphs, we will explain which aspects of Infinispan you're likely to want to reconfigure from their defaults. All attributes and elements from Infinispan which we don't mention are safe to ignore. Refer to the *Infinispan User Guide* [http://infinispan.org/documentation/] for the guru level performance tuning and customizations.

An Infinispan configuration file is an XML file complying with the Infinispan schema; the basic structure is shown in the following example:

Example 9.1. Simple structure of an infinispan xml configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:infinispan:config:7.0 http://www.infinispan.org/schemas/infinispan-
config-7.0.xsd"
    xmlns="urn:infinispan:config:7.0">

    <cache-container name="HibernateOGM" default-cache="DEFAULT">

        <!-- *************************** -->
        <!--   Default cache settings    -->
        <!-- *************************** -->
        <local-cache name="DEFAULT">
            <transaction mode="NON_DURABLE_XA"
                                                                            transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup"/>
        </local-cache>

        <local-cache name="User"/>

        <local-cache name="Order"/>

        <local-cache name="associations_User_Order"/>
    </cache-container>
</infinispan>
```

There are global settings that can be set before the `cache_container` section. These settings will affect the whole instance; mainly of interest for Hibernate OGM users is the `jgroups` element in which we will set JGroups configuration overrides.

Inside the `cache-container` section are defined explicit named caches and their configurations as well as the default cache (named `DEFAULT` here) if we want to affect all named caches. This is where we will likely want to configure clustering modes, eviction policies and `CacheStores`.

9.2. Manage data size

In its default configuration Infinispan stores all data in the heap of the JVM; in this barebone mode it is conceptually not very different than using a HashMap: the size of the data should fit in the heap of your VM, and stopping/killing/crashing your application will get all data lost with no way to recover it.

To store data permanently (out of the JVM memory) a `CacheStore` should be enabled. The `infinispan-core.jar` includes a simple implementation able to store data in simple binary files, on any read/write mounted filesystem; this is an easy starting point, but the real stuff is to be found in the additional modules found in the Infinispan distribution. Here you can find many more implementations to store your data in anything from JDBC connected relational databases, other NoSQL engines, to cloud storage services or other Infinispan clusters. Finally, implementing a custom `CacheStore` is quite easy.

To limit the memory consumption of the precious heap space, you can activate a `passivation` or an `eviction` policy; again there are several strategies to play with, for now let's just consider you'll likely need one to avoid running out of memory when storing too many entries in the bounded JVM memory space; of course you don't need to choose one while experimenting with limited data sizes: enabling such a strategy doesn't have any other impact in the functionality of your Hibernate OGM application (other than performance: entries stored in the Infinispan in-memory space is accessed much quicker than from any CacheStore).

A `CacheStore` can be configured as write-through, committing all changes to the `CacheStore` before returning (and in the same transaction) or as write-behind. A write-behind configuration is normally not encouraged in storage engines, as a failure of the node implies some data might be lost without receiving any notification about it, but this problem is mitigated in Infinispan because of its capability to combine CacheStore write-behind with a synchronous replication to other Infinispan nodes.

Example 9.2. Enabling a FileCacheStore and eviction

```
<local-cache name="User">
    <transaction mode="NON_DURABLE_XA"
                                                                           transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup"/>
```

```
    <eviction strategy="LIRS" max-entries="2000"/>
    <persistence passivation="true">
        <file-store
           shared="false"
           path="/var/infinispan/myapp/users"
            <write-behind flush-lock-timeout="15000" thread-pool-size="5" />
        </file-store>
    </persistence>
</local-cache>
```

In this example we enabled both `eviction` and a `CacheStore` (the `persistence` element). `LIRS` is one of the choices we have for eviction strategies. Here it is configured to keep (approximately) 2000 entries in live memory and evict the remaining as a memory usage control strategy.

The `CacheStore` is enabling `passivation`, which means that the entries which are evicted are stored on the filesystem.

Warning

You could configure an eviction strategy while not configuring a passivating CacheStore! That is a valid configuration for Infinispan but will have the evictor permanently remove entries. Hibernate OGM will break in such a configuration.

9.3. Clustering: store data on multiple Infinispan nodes

The best thing about Infinispan is that all nodes are treated equally and it requires almost no beforehand capacity planning: to add more nodes to the cluster you just have to start new JVMs, on the same or different physical servers, having your same Infinispan configuration and your same application.

Infinispan supports several clustering *cache modes*; each mode provides the same API and functionality but with different performance, scalability and availability options:

Infinispan cache modes

local
: Useful for a single VM: networking stack is disabled

replication
: All data is replicated to each node; each node contains a full copy of all entries. Consequentially reads are faster but writes don't scale as well. Not suited for very large datasets.

distribution
: Each entry is distributed on multiple nodes for redundancy and failure recovery, but not to all the nodes. Provides linear scalability for both write and read operations. distribution is the default mode.

To use the `replication` or `distribution` cache modes Infinispan will use JGroups to discover and connect to the other nodes.

In the default configuration, JGroups will attempt to autodetect peer nodes using a multicast socket; this works out of the box in the most network environments but will require some extra configuration in cloud environments (which often block multicast packets) or in case of strict firewalls. See the *JGroups reference documentation* [http://www.jgroups.org/manual/html_single/], specifically look for *Discovery Protocols* to customize the detection of peer nodes.

Nowadays, the `JVM` defaults to use `IPv6` network stack; this will work fine with JGroups, but only if you configured `IPv6` correctly. It is often useful to force the `JVM` to use `IPv4`.

It is also useful to let JGroups know which networking interface you want to use; especially if you have multiple interfaces it might not guess correctly.

Example 9.3. JVM properties to set for clustering

```
#192.168.122.1 is an example IPv4 address
-Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=192.168.122.1
```

Note

You don't need to use `IPv4`: JGroups is compatible with `IPv6` provided you have routing properly configured and valid addresses assigned.

The `jgroups.bind_addr` needs to match a placeholder name in your JGroups configuration in case you don't use the default one.

The default configuration uses `distribution` as cache mode and uses the `jgroups-tcp.xml` configuration for JGroups, which is contained in the Infinispan jar as the default configuration for Infinispan users. Let's see how to reconfigure this:

Example 9.4. Reconfiguring cache mode and override JGroups configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:infinispan:config:7.0 http://www.infinispan.org/schemas/infinispan-
config-7.0.xsd"
    xmlns="urn:infinispan:config:7.0">

    <jgroups>
        <stack-file name="custom-stack" path="my-jgroups-conf.xml" />
    </jgroups>

    <cache-container name="HibernateOGM" default-cache="DEFAULT">
        <transport stack="custom-stack" />
        <!-- *************************************** -->
        <!--     Default cache used as template      -->
        <!-- *************************************** -->
        <distrubuted-cache name="DEFAULT" mode="SYNC">
            <locking striping="false" acquire-timeout="10000"
                concurrency-level="500" write-skew="false" />
            <transaction mode="NON_DURABLE_XA"
                                                                           transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" />
            <state-transfer enabled="true" timeout="480000"
                await-initial-transfer="true" />
        </distributed-cache>

        <!-- Override the cache mode: -->
        <replicated-cache name="User" mode="SYNC">
            <locking striping="false" acquire-timeout="10000"
                concurrency-level="500" write-skew="false" />
            <transaction mode="NON_DURABLE_XA"
                                                                           transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" />
            <state-transfer enabled="true" timeout="480000"
                await-initial-transfer="true" />
        </replicated-cache>

        <distributed-cache name="Order" mode="SYNC">
            <locking striping="false" acquire-timeout="10000"
                concurrency-level="500" write-skew="false" />
            <transaction mode="NON_DURABLE_XA"
                                                                           transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" />
            <state-transfer enabled="true" timeout="480000"
                await-initial-transfer="true" />
        </distributed-cache>

        <distributed-cache name="associations_User_Order" mode="SYNC">
            <locking striping="false" acquire-timeout="10000"
                concurrency-level="500" write-skew="false" />
            <transaction mode="NON_DURABLE_XA"
                                                                           transaction-manager-
lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" />
            <state-transfer enabled="true" timeout="480000"
                await-initial-transfer="true" />
        </distributed-cache>

    </cache-container>

</infinispan>
```

In the example above we specify a custom JGroups configuration file and set the cache mode for the default cache to `distribution`; this is going to be inherited by the `Order` and the `associations_User_Order` caches. But for `User` we have chosen (for the sake of this example) to use `replication`.

Now that you have clustering configured, start the service on multiple nodes. Each node will need the same configuration and jars.

Tip

We have just shown how to override the clustering mode and the networking stack for the sake of completeness, but you don't have to!

Start with the default configuration and see if that fits you. You can fine tune these setting when you are closer to going in production.

9.4. Storage principles

To describe things simply, each entity is stored under a single key. The value itself is a map containing the columns / values pair.

Each association from one entity instance to (a set of) another is stored under a single key. The value contains the navigational information to the (set of) entity.

9.4.1. Properties and built-in types

Each entity is represented by a map. Each property or more precisely column is represented by an entry in this map, the key being the column name.

Hibernate OGM support by default the following property types:

- `java.lang.String`
- `java.lang.Character` (or char primitive)
- `java.lang.Boolean` (or boolean primitive)
- `java.lang.Byte` (or byte primitive)
- `java.lang.Short` (or short primitive)
- `java.lang.Integer` (or integer primitive)
- `java.lang.Long` (or long primitive)
- `java.lang.Integer` (or integer primitive)
- `java.lang.Float` (or float primitive)
- `java.lang.Double` (or double primitive)
- `java.math.BigDecimal`
- `java.math.BigInteger`

- `java.util.Calendar`
- `java.util.Date`
- `java.util.UUID`
- `java.util.URL`

Note

Hibernate OGM doesn't store null values in Infinispan, setting a value to null is the same as removing the corresponding entry from Infinispan.

This can have consequences when it comes to queries on null value.

9.4.2. Identifiers

Entity identifiers are used to build the key in which the entity is stored in the cache.

The key is comprised of the following information:

- the identifier column names
- the identifier column values
- the entity table (for the `CACHE_PER_KIND` strategy)

In `CACHE_PER_TABLE`, the table name is inferred from the cache name. In `CACHE_PER_KIND`, the table name is necessary to identify the entity in the generic cache.

Example 9.5. Define an identifier as a primitive type

```
@Entity
public class Bookmark {

    @Id
    private Long id;

    private String title;

    // getters, setters ...
}
```

Table 9.1. Content of the `Bookmark` cache in `CACHE_PER_TABLE`

KEY	MAP ENTRIES	
["id"], [42]	id	42

KEY	MAP ENTRIES	
	title	"Hibernate OGM documentation"

Table 9.2. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"Bookmark", ["id"], [42]	id	42
	title	"Hibernate OGM documentation"

Example 9.6. Define an identifier using @EmbeddedId

```
@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}

@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;
    private String content;

    // getters, setters ...
}
```

Table 9.3. Content of the `News` cache in `CACHE_PER_TABLE`

KEY	MAP ENTRIES	
[newsId.author, newsId.title], ["Guillaume", "How to use Hibernate OGM ?"]	newsId.author	"Guillaume"
	newsId.title	"How to use Hibernate OGM ?"
	content	"Simple, just like ORM but with a NoSQL database"

Table 9.4. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"News", [newsId.author, newsId.title],	newsId.author	"Guillaume"

KEY	MAP ENTRIES	
["Guillaume", "How to use Hibernate OGM ?"]	newsId.title	"How to use Hibernate OGM ?"
	content	"Simple, just like ORM but with a NoSQL database"

9.4.2.1. Identifier generation strategies

Since Infinispan has not native sequence nor identity column support, these are simulated using the table strategy, however their default values vary. We highly recommend you explicitly use a `TABLE` strategy if you want to generate a monotonic identifier.

But if you can, use a pure in-memory and scalable strategy like a UUID generator.

Example 9.7. Id generation strategy TABLE using default values

```
@Entity
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long id;

    private String name;

    // getters, setters ...
}
```

Table 9.5. Content of the `hibernate_sequences` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["sequence_name"], ["default"]	2

Table 9.6. Content of the IDENTIFIERS cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"hibernate_sequences", ["sequence_name"], ["default"]	2

As you can see, in `CACHE_PER_TABLE`, the key does not contain the id source table name. It is inferred by the cache name hosting that key.

Example 9.8. Id generation strategy TABLE using a custom table

```
@Entity
```

```
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "guitarGen")
    @TableGenerator(
        name = "guitarGen",
        table = "GuitarPlayerSequence",
        pkColumnName = "seq"
        pkColumnValue = "guitarPlayer",
    )
    private long id;

    // getters, setters ...
}
```

Table 9.7. Content of the `GuitarPlayerSequence` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["seq"], ["guitarPlayer"]	2

Table 9.8. Content of the IDENTIFIERS cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"GuitarPlayerSequence", ["seq"], ["guitarPlayer"]	2

Example 9.9. SEQUENCE id generation strategy

```
@Entity
public class Song {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "songSequenceGenerator")
  @SequenceGenerator(
      name = "songSequenceGenerator",
      sequenceName = "song_sequence",
      initialValue = 2,
      allocationSize = 20
  )
  private Long id;

  private String title;

  // getters, setters ...
}
```

Table 9.9. Content of the `hibernate_sequences` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["sequence_name"], ["song_sequence"]	11

Table 9.10. Content of the `IDENTIFIERS` cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"hibernate_sequences", "["sequence_name"], ["song_sequence"]	11

9.4.3. Entities

Entities are stored in the cache named after the entity name when using the `CACHE_PER_TABLE` strategy. In the `CACHE_PER_KIND` strategy, entities are stored in a single cache named `ENTITIES`.

The key is comprised of the following information:

- the identifier column names
- the identifier column values
- the entity table (for the `CACHE_PER_KIND` strategy)

In `CACHE_PER_TABLE`, the table name is inferred from the cache name. In `CACHE_PER_KIND`, the table name is necessary to identify the entity in the generic cache.

The entry value is an instance of `org.infinispan.atomic.FineGrainedMap` which contains all the entity properties - or to be specific columns. Each column name and value is stored as a key / value pair in the map. We use this specialized map as Infinispan is able to transport changes in a much more efficient way.

Example 9.10. Default JPA mapping for an entity

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    // getters, setters ...
}
```

Table 9.11. Content of the `News` cache in `CACHE_PER_TYPE`

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	title	"On the merits of NoSQL"

Table 9.12. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"News", ["id"], ["1234-5678"]	id	"1234-5678"
	title	"On the merits of NoSQL"

As you can see, the table name is not part of the key for `CACHE_PER_TYPE`. In the rest of this section we will no longer show the `CACHE_PER_KIND` strategy.

Example 9.11. Rename field and collection using @Table and @Column

```
@Entity
@Table(name = "Article")
public class News {

    @Id
    private String id;

    @Column(name = "headline")
    private String title;

    // getters, setters ...
}
```

Table 9.13. Content of the `Article` cache

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	headline	"On the merits of NoSQL"

9.4.3.1. Embedded objects and collections

Example 9.12. Embedded object

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    @Embedded
    private NewsPaper paper;

    // getters, setters ...
}

@Embeddable
public class NewsPaper {
    private String name;
    private String owner;

    // getters, setters ...
}
```

Table 9.14. Content of the `News` cache

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	title	"On the merits of NoSQL"
	paper.name	"NoSQL journal of prophecies"
	paper.owner	"Delphy"

Example 9.13. @ElementCollection with one attribute

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

Table 9.15. Content of the `GrandMother` cache

KEY	MAP ENTRIES	
["id"], ["granny"]	id	"granny"

Table 9.16. Content of the `associations_GrandMother_grandChildren` cache in `CACHE_PER_TYPE`

KEY	ROW KEY	ROW MAP ENTRIES	
["GrandMother_id"], ["granny"]		GrandMother_id	"granny"

KEY	ROW KEY	ROW MAP ENTRIES	
	["GrandMother_id", "name"], ["granny", "Leia"]	name	"Leia"
	["GrandMother_id", "name"], ["granny", "Luke"]	GrandMother_id	"granny"
		name	"Luke"

Table 9.17. Content of the `ASSOCIATIONS` cache in `CACHE_PER_KIND`

KEY	ROW KEY	ROW MAP ENTRIES	
"GrandMother_grandChildren", ["GrandMother_id"], ["granny"]	["GrandMother_id", "name"], ["granny", "Leia"]	GrandMother_id	"granny"
		name	"Leia"
	["GrandMother_id", "name"], ["granny", "Luke"]	GrandMother_id	"granny"
		name	"Luke"

Here, we see that the collection of elements is stored in a separate cache and entry. The association key is made of:

- the foreign key column names pointing to the owner of this association
- the foreign key column values pointing to the owner of this association
- the association table name in the `CACHE_PER_KIND` approach where all associations share the same cache

The association entry is a map containing the representation of each entry in the collection. The keys of that map are made of:

- the names of the columns uniquely identifying that specific collection entry (e.g. for a `Set` this is all of the columns)
- the values of the columns uniquely identifying that specific collection entry

The value attack to that collection entry key is a Map containing the key value pairs column name / column value.

Example 9.14. @ElementCollection with @OrderColumn

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    @OrderColumn( name = "birth_order" )
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

Table 9.18. Content of the `GrandMother` cache

KEY	MAP ENTRIES	
["id"], ["granny"]	id	"granny"

Table 9.19. Content of the `GrandMother_grandChildren` cache

KEY	ROW KEY	ROW MAP ENTRIES	
["GrandMother_id"], ["granny"]	["GrandMother_id", "birth_order"], ["granny", 0]	GrandMother_id	"granny"
		birth_order	0
		name	"Leia"
	["GrandMother_id", "birth_order"], ["granny", 1]	GrandMother_id	"granny"
		birth_order	1
		name	"Luke"

Here we used an indexed collection and to identify the entry in the collection, only the owning entity id and the index value is enough.

9.4.4. Associations

Associations between entities are mapped like (collection of) embeddables except that the target entity is represented by its identifier(s).

Example 9.15. Unidirectional one-to-one

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}
```

```
@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    private Vehicule vehicule;

    // getters, setters ...
}
```

Table 9.20. Content of the `Vehicule` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"
	brand	"Mercedes"

Table 9.21. Content of the `Wheel` cache

KEY	MAP ENTRIES	
["id"], ["W001"]	id	"W001"
	diameter	0.0
	vehicule_id	"V_01"

Example 9.16. Unidirectional one-to-one with @JoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @JoinColumn( name = "part_of" )
    private Vehicule vehicule;
    // getters, setters ...
}
```

Table 9.22. Content of the `Vehicle` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"
	brand	"Mercedes"

Table 9.23. Content of the `Wheel` cache

KEY	MAP ENTRIES	
"Wheel", ["id"], ["W001"]	id	"W001"
	diameter	0.0
	part_of	"V_01"

Example 9.17. Unidirectional one-to-one with @MapsId and @PrimaryKeyJoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}

@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @PrimaryKeyJoinColumn
    @MapsId
    private Vehicule vehicule;

    // getters, setters ...
}
```

Table 9.24. Content of the `Vehicle` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"

KEY	MAP ENTRIES	
	brand	"Mercedes"

Table 9.25. Content of the `Wheel` cache

KEY	MAP ENTRIES	
["vehicule_id"], ["V_01"]	vehicule_id	"V_01"
	diameter	0.0

Example 9.18. Bidirectional one-to-one

```
@Entity
public class Husband {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Wife wife;

    // getters, setters ...
}

@Entity
public class Wife {

    @Id
    private String id;
    private String name;

    @OneToOne(mappedBy="wife")
    private Husband husband;

    // getters, setters ...
}
```

Table 9.26. Content of the `Husband` cache

KEY	MAP ENTRIES	
["id"], ["alex"]	id	"alex"
	name	"Alex"
	wife	"bea"

Table 9.27. Content of the `Wife` cache

KEY	MAP ENTRIES	
["id"], ["bea"]	id	"bea"

KEY	MAP ENTRIES	
	name	"Bea"

Table 9.28. Content of the `associations_Husband` cache

KEY	ROW KEY	MAP ENTRIES	
["wife"], ["bea"]	["id", "wife"], ["alex", "bea"]	id	"alex"
		wife	"bea"

Example 9.19. Unidirectional one-to-many

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Table 9.29. Content of the `Basket` cache

KEY	MAP ENTRIES	
["id"], ["davide_basket"]	id	"davide_basket"
	owner	"Davide"

Table 9.30. Content of the `Product` cache

KEY	MAP ENTRIES	
["name"], ["Beer"]	name	"Beer"
	description	"Tactical Nuclear Penguin"

KEY	MAP ENTRIES	
["name"], ["Pretzel"]	name	"Pretzel"
	description	"Glutino Pretzel Sticks"

Table 9.31. Content of the `associations_Basket_Product` cache

KEY	ROW KEY	MAP ENTRIES	
["Basket_id"], ["davide_basket"]	["Basket_id", "products_name"], ["davide_basket", "Beer"]	Basket_id	"davide_basket"
		products_name	"Beer"
	["Basket_id", "products_name"], ["davide_basket", "Pretzel"]	Basket_id	"davide_basket"
		products_name	"Pretzel"

Example 9.20. Unidirectional one-to-many with `@JoinTable`

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    @JoinTable( name = "BasketContent" )
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Table 9.32. Content of the `Basket` cache

KEY	MAP ENTRIES	
["id"], ["davide_basket"]	id	"davide_basket"

KEY	MAP ENTRIES	
	owner	"Davide"

Table 9.33. Content of the `Basket` cache

KEY	MAP ENTRIES	
["name"], ["Beer"]	name	"Beer"
	description	"Tactical Nuclear Penguin"
["name"], ["Pretzel"]	name	"Pretzel"
	description	"Glutino Pretzel Sticks"

Table 9.34. Content of the `associations_BasketContent` cache

KEY	ROW KEY	MAP ENTRIES	
["Basket_id"], ["davide_basket"]	["Basket_id", "products_name"], ["davide_basket", "Beer"]	Basket_id	"davide_basket"
		products_name	"Beer"
	["Basket_id", "products_name"], ["davide_basket", "Pretzel"]	Basket_id	"davide_basket"
		products_name	"Pretzel"

Example 9.21. Unidirectional one-to-many using maps with defaults

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

Table 9.35. Content of the `User` cache

KEY	MAP ENTRIES	
["id"], ["user_001"]	id	"user_001"

Table 9.36. Content of the `Address` cache

KEY	MAP ENTRIES	
["id"], ["address_001"]	id	"address_001"
	city	"Rome"
["id"], ["address_002"]	id	"address_002"
	city	"Paris"

Table 9.37. Content of the `associations_User_address` cache

KEY	ROW KEY	MAP ENTRIES	
["User_id"], "user_001"]	["User_id", "addresses_KEY"], ["user_001", "home"]	User_id	"user_001"
		addresses_KEY	"home"
		addresses_id	"address_001"
	["User_id", "addresses_KEY"], ["user_001", "work"]	User_id	"user_002"
		addresses_KEY	"work"
		addresses_id	"address_002"

Example 9.22. Unidirectional one-to-many using maps with @MapKeyColumn

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    @MapKeyColumn(name = "addressType")
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;
    // getters, setters ...
}
```

Table 9.38. Content of the `User` cache

KEY	MAP ENTRIES	
["id"], ["user_001"]	id	"user_001"

Table 9.39. Content of the `Address` cache

KEY	MAP ENTRIES	
["id"], ["address_001"]	id	"address_001"
	city	"Rome"
["id"], ["address_002"]	id	"address_002"
	city	"Paris"

Table 9.40. Content of the `associations_User_address` cache

KEY	ROW KEY	MAP ENTRIES	
["User_id"], "user_001"]	["User_id", "addressType"], ["user_001", "home"]	User_id	"user_001"
		addressesType	"home"
		addresses_id	"address_001"
	["User_id", "addressType"], ["user_001", "work"]	User_id	"user_002"
		addressesType	"work"
		addresses_id	"address_002"

Example 9.23. Unidirectional many-to-one

```
@Entity
public class JavaUserGroup {

    @Id
    private String jugId;
    private String name;

    // getters, setters ...
}

@Entity
public class Member {

    @Id
    private String id;
    private String name;
    @ManyToOne
    private JavaUserGroup memberOf;

    // getters, setters ...
}
```

Table 9.41. Content of the `JavaUserGroup` cache

KEY	MAP ENTRIES	
["jugId"], ["summer_camp"]	jugId	"summer_camp"
	name	"JUG Summer Camp"

Table 9.42. Content of the `Member` cache

KEY	MAP ENTRIES	
["member_id"], ["emmanuel"]	member_id	"emmanuel"
	name	"Emmanuel Bernard"
	memberOf_jug_id	"summer_camp"
["member_id"], ["jerome"]	member_id	"jerome"
	name	"Jerome"
	memberOf_jug_id	"summer_camp"

Example 9.24. Bidirectional many-to-one

```
@Entity
public class SalesForce {

    @Id
    private String id;
    private String corporation;

    @OneToMany(mappedBy = "salesForce")
    private Set<SalesGuy> salesGuys = new HashSet<SalesGuy>();

    // getters, setters ...
}

@Entity
public class SalesGuy {
    private String id;
    private String name;

    @ManyToOne
    private SalesForce salesForce;

    // getters, setters ...
}
```

Table 9.43. Content of the `SalesForce` cache

KEY	MAP ENTRIES	
["id"], ["red_hat"]	id	"red_hat"
	corporation	"Red Hat"

Table 9.44. Content of the `SalesGuy` cache

KEY	MAP ENTRIES	
["id"], ["eric"]	id	"eric"
	name	"Eric"
	salesForce_id	"red_hat"
["id"], ["simon"]	id	"simon"
	name	"Simon"
	salesForce_id	"red_hat"

Table 9.45. Content of the `associations_SalesGuy` cache

KEY	ROW KEY	MAP ENTRIES	
["salesForce_id"], ["red_hat"]	["salesForce_id", "id"], ["red_hat", "eric"]	salesForce_id	"red_hat"
		id	"eric"
	["salesForce_id", "id"], ["red_hat", "simon"]	salesForce_id	"red_hat"
		id	"simon"

Example 9.25. Unidirectional many-to-many

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();
    // getters, setters ...
}
```

The "Math" class has 2 students: John Doe and Mario Rossi

The "English" class has 2 students: Kate Doe and Mario Rossi

Table 9.46. Content of the `ClassRoom` cache

KEY	MAP ENTRIES	
["id"], [1]	id	1
	name	"Math"
["id"], [2]	id	2
	name	"English"

Table 9.47. Content of the `Student` cache

KEY	MAP ENTRIES	
["id"], ["john"]	id	"john"
	name	"John Doe"
["id"], ["mario"]	id	"mario"
	name	"Mario Rossi"
["id"], ["kate"]	id	"kate"
	name	"Kate Doe"

Table 9.48. Content of the `associations_ClassRoom_Student` cache

KEY	ROW KEY	MAP ENTRIES	
["ClassRoom_id"], [1]	["ClassRoom_id", "students_id"], [1, "mario"]	ClassRoom_id	1
		students_id	"mario"
	["ClassRoom_id", "students_id"], [1, "john"]	ClassRoom_id	1
		students_id	"john"
["ClassRoom_id"], [2]	["ClassRoom_id", "students_id"], [2, "kate"]	ClassRoom_id	2
		students_id	"kate"
	["ClassRoom_id", "students_id"], [2, "mario"]	ClassRoom_id	2
		students_id	"mario"

Example 9.26. Bidirectional many-to-many

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany( mappedBy = "bankAccounts" )
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

David owns 2 accounts: "012345" and "ZZZ-009"

Table 9.49. Content of the `AccountOwner` cache

KEY	MAP ENTRIES	
["id"], ["David"]	id	"David"
	SSN	"0123456"

Table 9.50. Content of the `BankAccount` cache

KEY	MAP ENTRIES	
["id"], ["account_1"]	id	"account_1"
	accountNumber	"X2345000"
["id"], ["account_2"]	id	"account_2"
	accountNumber	"ZZZ-009"

Table 9.51. Content of the `AccountOwner_BankAccount` cache

KEY	ROW KEY	MAP ENTRIES	
["bankAccounts_id"], ["account_1"]	["bankAccounts_id", "owners_id"], ["account_1", "David"]	bankAccounts_id	"account_1"
		owners_id	"David"
["bankAccounts_id"], ["account_2"]	["bankAccounts_id", "owners_id"], ["account_2", "David"]	bankAccounts_id	"account_2"
		owners_id	"David"
["owners_id"], ["David"]	["owners_id", "banksAccounts_id"], ["David", "account_1"]	bankAccounts_id	"account_1"
		owners_id	"David"
	["owners_id", "banksAccounts_id"], ["David", "account_2"]	bankAccounts_id	"account_2"
		owners_id	"David"

9.5. Transactions

Infinispan supports transactions and integrates with any standard JTA `TransactionManager`; this is a great advantage for JPA users as it allows to experience a *similar* behaviour to the one we are used to when we work with RDBMS databases.

If you're having Hibernate OGM start and manage Infinispan, you can skip this as it will inject the same `TransactionManager` instance which you already have set up in the Hibernate / JPA configuration.

If you are providing an already started Infinispan CacheManager instance by using the `JNDI` lookup approach, then you have to make sure the CacheManager is using the same `TransactionManager` as Hibernate:

Example 9.27. Configuring a JBoss Standalone TransactionManager lookup in Infinispan configuration

```
<default>
   <transaction
      transactionMode="TRANSACTIONAL"
      transactionManagerLookupClass=
    "org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" />
</default>
```

Infinispan supports different transaction modes like `PESSIMISTIC` and `OPTIMISTIC`, supports `XA` recovery and provides many more configuration options; see the *Infinispan User Guide* [http://infinispan.org/documentation/] for more advanced configuration options.

9.6. Storing a Lucene index in Infinispan

Hibernate Search, which can be used for advanced query capabilities (see *Chapter 7, Query your entities*), needs some place to store the indexes for its embedded `Apache Lucene` engine.

A common place to store these indexes is the filesystem which is the default for Hibernate Search; however if your goal is to scale your NoSQL engine on multiple nodes you need to share this index. Network sharing file systems are a possibility but we don’t recommended that. Often the best option is to store the index in whatever NoSQL database you are using (or a different dedicated one).

Tip

You might find this section useful even if you don’t intend to store your data in Infinispan.

The Infinispan project provides an adaptor to plug into Apache Lucene, so that it writes the indexes in Infinispan and searches data in it. Since Infinispan can be used as an application cache to other NoSQL storage engines by using a CacheStore (see *Section 9.2, “Manage data size”*) you can use this adaptor to store the Lucene indexes in any NoSQL store supported by Infinispan:

- Cassandra
- Filesystem (but locked correctly at the Infinispan level)
- MongoDB
- HBase
- JDBC databases
- JDBM
- BDBJE
- A secondary (independent) Infinispan grid
- Any Cloud storage service *supported by JClouds* [http://www.jclouds.org/documentation/reference/supported-providers/]

How to configure it? Here is a simple cheat sheet to get you started with this type of setup:

- Add `org.hibernate:hibernate-search-infinispan:5.1.0.Final` to your dependencies
- set these configuration properties:
  - `hibernate.search.default.directory_provider = infinispan`

- `hibernate.search.default.exclusive_index_use = false`
- `hibernate.search.infinispan.configuration_resourcename = [infinispan configuration filename]`

The referenced Infinispan configuration should define a `CacheStore` to load/store the index in the NoSQL engine of choice. It should also define three cache names:

Table 9.52. Infinispan caches used to store indexes

Cache name	Description	Suggested cluster mode
LuceneIndexesLocking	Transfers locking information. Does not need a cache store.	replication
LuceneIndexesData	Contains the bulk of Lucene data. Needs a cache store.	distribution + L1
LuceneIndexesMetadata	Stores metadata on the index segments. Needs a cache store.	replication

This configuration is not going to scale well on write operations: to do that you should read about the master/slave and sharding options in Hibernate Search. The complete explanation and configuration options can be found in the *Hibernate Search Reference Guide* [http://docs.jboss.org/hibernate/search/4.2/reference/en-US/html_single/#infinispan-directories]

Some NoSQL support storage of Lucene indexes directly, in which case you might skip the Infinispan Lucene integration by implementing a custom `DirectoryProvider` for Hibernate Search. You're very welcome to share the code and have it merged in Hibernate Search for others to use, inspect, improve and maintain.

Ehcache

When combined with Hibernate ORM, Ehcache is commonly used as a 2nd level cache to cache data whose primary storage is a relational database. When used with Hibernate OGM it is not "just a cache" but is the main storage engine for your data.

This is not the reference manual for Ehcache itself: we're going to list only how Hibernate OGM should be configured to use Ehcache; for all the tuning and advanced options please refer to the *Ehcache Documentation* [http://www.ehcache.org/documentation].

10.1. Configure Ehcache

Two steps:

- Add the dependencies to classpath
- And then choose one of:
  - Use the default Ehcache configuration (no action needed)
  - Point to your own configuration resource name

10.1.1. Adding Ehcache dependencies

To add the dependencies via some Maven-definitions-using tool, add the following module:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-ehcache</artifactId>
    <version>4.1.3.Final</version>
</dependency>
```

If you're not using a dependency management tool, copy all the dependencies from the distribution in the directories:

- `/lib/required`
- `/lib/ehcache`
- Optionally - depending on your container - you might need some of the jars from `/lib/provided`

10.1.2. Ehcache specific configuration properties

Hibernate OGM expects you to define an Ehcache configuration in its own configuration resource; all what we need to set it the resource name.

To use the default configuration provided by Hibernate OGM - which is a good starting point for new users - you don't have to set any property.

Ehcache datastore configuration properties

hibernate.ogm.datastore.provider
: To use Ehcache as a datastore provider set it to `ehcache`.

hibernate.ogm.ehcache.configuration_resource_name
: Should point to the resource name of an Ehcache configuration file. Defaults to `org/hibernate/ogm/datastore/ehcache/default-ehcache.xml`.

`hibernate.ogm.datastore.keyvalue.cache_storage`
: The strategy for persisting data in EhCache. The following two strategies exist (values of the `org.hibernate.ogm.datastore.keyvalue.options.CacheMappingType` enum):

  - `CACHE_PER_TABLE`: A dedicated cache will be used for each entity type, association type and id source table.
  - `CACHE_PER_KIND`: Three caches will be used: one cache for all entities, one cache for all associations and one cache for all id sources.

  Defaults to `CACHE_PER_TABLE`. It is the recommended strategy as it makes it easier to target a specific cache for a given entity.

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `EhcacheProperties` when specifying the configuration properties listed above.

Common properties shared between stores are declared on `OgmProperties` (a super interface of `EhcacheProperties`).

For maximum portability between stores, use the most generic interface possible.

10.1.3. Cache names used by Hibernate OGM

Depending on the cache mapping approach, Hibernate OGM will either:

- store each entity type, association type and id source table in a dedicated cache very much like what Hibernate ORM would do. This is the `CACHE_PER_TABLE` approach.
- store data in three different caches when using the `CACHE_PER_KIND` approach:

- `ENTITIES`: is going to be used to store the main attributes of all your entities.
- `ASSOCIATIONS`: stores the association information representing the links between entities.
- `IDENTIFIER_STORE`: contains internal metadata that Hibernate OGM needs to provide sequences and auto-incremental numbers for primary key generation.

The preferred strategy is `CACHE_PER_TABLE` as it offers both more fine grained configuration options and the ability to work on specific entities in a more simple fashion.

10.2. Storage principles

To describe things simply, each entity is stored under a single key. The value itself is a map containing the columns / values pair.

Each association from one entity instance to (a set of) another is stored under a single key. The value contains the navigational information to the (set of) entity.

10.2.1. Properties and built-in types

Each entity is represented by a map. Each property or more precisely column is represented by an entry in this map, the key being the column name.

Hibernate OGM supports by default the following property types:

- `java.lang.String`
- `java.lang.Character` (or char primitive)
- `java.lang.Boolean` (or boolean primitive)
- `java.lang.Byte` (or byte primitive)
- `java.lang.Short` (or short primitive)
- `java.lang.Integer` (or integer primitive)
- `java.lang.Long` (or long primitive)
- `java.lang.Integer` (or integer primitive)
- `java.lang.Float` (or float primitive)
- `java.lang.Double` (or double primitive)
- `java.math.BigDecimal`
- `java.math.BigInteger`

- `java.util.Calendar`
- `java.util.Date`
- `java.util.UUID`
- `java.util.URL`

Note

Hibernate OGM doesn't store null values in Ehcache, setting a value to null is the same as removing the corresponding entry from Ehcache.

This can have consequences when it comes to queries on null value.

10.2.2. Identifiers

Entity identifiers are used to build the key in which the entity is stored in the cache.

The key is comprised of the following information:

- the identifier column names
- the identifier column values
- the entity table (for the `CACHE_PER_KIND` strategy)

In `CACHE_PER_TABLE`, the table name is inferred from the cache name. In `CACHE_PER_KIND`, the table name is necessary to identify the entity in the generic cache.

Example 10.1. Define an identifier as a primitive type

```
@Entity
public class Bookmark {

    @Id
    private Long id;

    private String title;

    // getters, setters ...
}
```

Table 10.1. Content of the `Bookmark` cache in `CACHE_PER_TABLE`

KEY	MAP ENTRIES	
["id"], [42]	id	42

KEY	MAP ENTRIES	
	title	"Hibernate OGM documentation"

Table 10.2. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"Bookmark", ["id"], [42]	id	42
	title	"Hibernate OGM documentation"

Example 10.2. Define an identifier using @EmbeddedId

```
@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}

@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;
    private String content;

    // getters, setters ...
}
```

Table 10.3. Content of the `News` cache in `CACHE_PER_TABLE`

KEY	MAP ENTRIES	
[newsId.author, newsId.title], ["Guillaume", "How to use Hibernate OGM ?"]	newsId.author	"Guillaume"
	newsId.title	"How to use Hibernate OGM ?"
	content	"Simple, just like ORM but with a NoSQL database"

Table 10.4. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"News", [newsId.author, newsId.title],	newsId.author	"Guillaume"

KEY	MAP ENTRIES	
["Guillaume", "How to use Hibernate OGM ?"]	newsId.title	"How to use Hibernate OGM ?"
	content	"Simple, just like ORM but with a NoSQL database"

10.2.2.1. Identifier generation strategies

Since Ehcache has not native sequence nor identity column support, these are simulated using the table strategy, however their default values vary. We highly recommend you explicitly use a `TABLE` strategy if you want to generate a monotonic identifier.

But if you can, use a pure in-memory and scalable strategy like a UUID generator.

Example 10.3. Id generation strategy TABLE using default values

```
@Entity
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long id;

    private String name;

    // getters, setters ...
}
```

Table 10.5. Content of the `hibernate_sequences` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["sequence_name"], ["default"]	2

Table 10.6. Content of the IDENTIFIERS cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"hibernate_sequences", ["sequence_name"], ["default"]	2

As you can see, in `CACHE_PER_TABLE`, the key does not contain the id source table name. It is inferred by the cache name hosting that key.

Example 10.4. Id generation strategy TABLE using a custom table

```
@Entity
```

```
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "guitarGen")
    @TableGenerator(
        name = "guitarGen",
        table = "GuitarPlayerSequence",
        pkColumnName = "seq"
        pkColumnValue = "guitarPlayer",
    )
    private long id;

    // getters, setters ...
}
```

Table 10.7. Content of the `GuitarPlayerSequence` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["seq"], ["guitarPlayer"]	2

Table 10.8. Content of the IDENTIFIERS cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"GuitarPlayerSequence", ["seq"], ["guitarPlayer"]	2

Example 10.5. SEQUENCE id generation strategy

```
@Entity
public class Song {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "songSequenceGenerator")
  @SequenceGenerator(
      name = "songSequenceGenerator",
      sequenceName = "song_sequence",
      initialValue = 2,
      allocationSize = 20
  )
  private Long id;

  private String title;

  // getters, setters ...
}
```

Table 10.9. Content of the `hibernate_sequences` cache in `CACHE_PER_TABLE`

KEY	NEXT VALUE
["sequence_name"], ["song_sequence"]	11

Table 10.10. Content of the `IDENTIFIERS` cache in `CACHE_PER_KIND`

KEY	NEXT VALUE
"hibernate_sequences", "["sequence_name"], ["song_sequence"]	11

10.2.3. Entities

Entities are stored in the cache named after the entity name when using the `CACHE_PER_TABLE` strategy. In the `CACHE_PER_KIND` strategy, entities are stored in a single cache named `ENTITIES`.

The key is comprised of the following information:

- the identifier column names
- the identifier column values
- the entity table (for the `CACHE_PER_KIND` strategy)

In `CACHE_PER_TABLE`, the table name is inferred from the cache name. In `CACHE_PER_KIND`, the table name is necessary to identify the entity in the generic cache.

The entry value is itself a map which contains all the entity properties - or to be specific columns. Each column name and value is stored as a key / value pair in the map.

Example 10.6. Default JPA mapping for an entity

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    // getters, setters ...
}
```

Table 10.11. Content of the `News` cache in `CACHE_PER_TYPE`

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	title	"On the merits of NoSQL"

Table 10.12. Content of the `ENTITIES` cache in `CACHE_PER_KIND`

KEY	MAP ENTRIES	
"News", ["id"], ["1234-5678"]	id	"1234-5678"

KEY	MAP ENTRIES	
	title	"On the merits of NoSQL"

As you can see, the table name is not part of the key for `CACHE_PER_TYPE`. In the rest of this section we will no longer show the `CACHE_PER_KIND` strategy.

Example 10.7. Rename field and collection using @Table and @Column

```
@Entity
@Table(name = "Article")
public class News {

    @Id
    private String id;

    @Column(name = "headline")
    private String title;

    // getters, setters ...
}
```

Table 10.13. Content of the `Article` cache

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	headline	"On the merits of NoSQL"

10.2.3.1. Embedded objects and collections

Example 10.8. Embedded object

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    @Embedded
    private NewsPaper paper;

    // getters, setters ...
}

@Embeddable
public class NewsPaper {

    private String name;
    private String owner;
    // getters, setters ...
}
```

Table 10.14. Content of the `News` cache

KEY	MAP ENTRIES	
["id"], ["1234-5678"]	id	"1234-5678"
	title	"On the merits of NoSQL"
	paper.name	"NoSQL journal of prophecies"
	paper.owner	"Delphy"

Example 10.9. @ElementCollection with one attribute

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

Table 10.15. Content of the `GrandMother` cache

KEY	MAP ENTRIES	
["id"], ["granny"]	id	"granny"

Table 10.16. Content of the `associations_GrandMother_grandChildren` cache in `CACHE_PER_TYPE`

KEY	ROW KEY	ROW MAP ENTRIES	
["GrandMother_id"], ["granny"]	["GrandMother_id", "name"], ["granny", "Leia"]	GrandMother_id	"granny"
		name	"Leia"

KEY	ROW KEY	ROW MAP ENTRIES	
	["GrandMother_id", "name"], ["granny", "Luke"]	GrandMother_id	"granny"
		name	"Luke"

Table 10.17. Content of the `ASSOCIATIONS` cache in `CACHE_PER_KIND`

KEY	ROW KEY	ROW MAP ENTRIES	
"GrandMother_grandChildren", ["GrandMother_id"], ["granny"]	["GrandMother_id", "name"], ["granny", "Leia"]	GrandMother_id	"granny"
		name	"Leia"
	["GrandMother_id", "name"], ["granny", "Luke"]	GrandMother_id	"granny"
		name	"Luke"

Here, we see that the collection of elements is stored in a separate cache and entry. The association key is made of:

- the foreign key column names pointing to the owner of this association
- the foreign key column values pointing to the owner of this association
- the association table name in the `CACHE_PER_KIND` approach where all associations share the same cache

The association entry is a map containing the representation of each entry in the collection. The keys of that map are made of:

- the names of the columns uniquely identifying that specific collection entry (e.g. for a `Set` this is all of the columns)
- the values of the columns uniquely identifying that specific collection entry

The value attack to that collection entry key is a Map containing the key value pairs column name / column value.

Example 10.10. @ElementCollection with @OrderColumn

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    @OrderColumn( name = "birth_order" )
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

Table 10.18. Content of the `GrandMother` cache

KEY	MAP ENTRIES	
["id"], ["granny"]	id	"granny"

Table 10.19. Content of the `GrandMother_grandChildren` cache

KEY	ROW KEY	ROW MAP ENTRIES	
["GrandMother_id"], ["granny"]	["GrandMother_id", "birth_order"], ["granny", 0]	GrandMother_id	"granny"
		birth_order	0
		name	"Leia"
	["GrandMother_id", "birth_order"], ["granny", 1]	GrandMother_id	"granny"
		birth_order	1
		name	"Luke"

Here we used an indexed collection and to identify the entry in the collection, only the owning entity id and the index value is enough.

10.2.4. Associations

Associations between entities are mapped like (collection of) embeddables except that the target entity is represented by its identifier(s).

Example 10.11. Unidirectional one-to-one

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}
```

```
@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    private Vehicule vehicule;

    // getters, setters ...
}
```

Table 10.20. Content of the `Vehicule` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"
	brand	"Mercedes"

Table 10.21. Content of the `Wheel` cache

KEY	MAP ENTRIES	
["id"], ["W001"]	id	"W001"
	diameter	0.0
	vehicule_id	"V_01"

Example 10.12. Unidirectional one-to-one with @JoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @JoinColumn( name = "part_of" )
    private Vehicule vehicule;
    // getters, setters ...
}
```

Table 10.22. Content of the `Vehicle` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"
	brand	"Mercedes"

Table 10.23. Content of the `Wheel` cache

KEY	MAP ENTRIES	
"Wheel", ["id"], ["W001"]	id	"W001"
	diameter	0.0
	part_of	"V_01"

Example 10.13. Unidirectional one-to-one with @MapsId and @PrimaryKeyJoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}

@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @PrimaryKeyJoinColumn
    @MapsId
    private Vehicule vehicule;

    // getters, setters ...
}
```

Table 10.24. Content of the `Vehicle` cache

KEY	MAP ENTRIES	
["id"], ["V_01"]	id	"V_01"

KEY	MAP ENTRIES	
	brand	"Mercedes"

Table 10.25. Content of the `Wheel` cache

KEY	MAP ENTRIES	
["vehicule_id"], ["V_01"]	vehicule_id	"V_01"
	diameter	0.0

Example 10.14. Bidirectional one-to-one

```
@Entity
public class Husband {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Wife wife;

    // getters, setters ...
}

@Entity
public class Wife {

    @Id
    private String id;
    private String name;

    @OneToOne(mappedBy="wife")
    private Husband husband;

    // getters, setters ...
}
```

Table 10.26. Content of the `Husband` cache

KEY	MAP ENTRIES	
["id"], ["alex"]	id	"alex"
	name	"Alex"
	wife	"bea"

Table 10.27. Content of the `Wife` cache

KEY	MAP ENTRIES	
["id"], ["bea"]	id	"bea"

KEY	MAP ENTRIES	
	name	"Bea"

Table 10.28. Content of the `associations_Husband` cache

KEY	ROW KEY	MAP ENTRIES	
["wife"], ["bea"]	["id", "wife"], ["alex", "bea"]	id	"alex"
		wife	"bea"

Example 10.15. Unidirectional one-to-many

```java
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Table 10.29. Content of the `Basket` cache

KEY	MAP ENTRIES	
["id"], ["davide_basket"]	id	"davide_basket"
	owner	"Davide"

Table 10.30. Content of the `Product` cache

KEY	MAP ENTRIES	
["name"], ["Beer"]	name	"Beer"
	description	"Tactical Nuclear Penguin"

KEY	MAP ENTRIES	
["name"], ["Pretzel"]	name	"Pretzel"
	description	"Glutino Pretzel Sticks"

Table 10.31. Content of the `associations_Basket_Product` cache

KEY	ROW KEY	MAP ENTRIES	
["Basket_id"], ["davide_basket"]	["Basket_id", "products_name"], ["davide_basket", "Beer"]	Basket_id	"davide_basket"
		products_name	"Beer"
	["Basket_id", "products_name"], ["davide_basket", "Pretzel"]	Basket_id	"davide_basket"
		products_name	"Pretzel"

Example 10.16. Unidirectional one-to-many with `@JoinTable`

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    @JoinTable( name = "BasketContent" )
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Table 10.32. Content of the `Basket` cache

KEY	MAP ENTRIES	
["id"], ["davide_basket"]	id	"davide_basket"

KEY	MAP ENTRIES	
	owner	"Davide"

Table 10.33. Content of the `Basket` cache

KEY	MAP ENTRIES	
["name"], ["Beer"]	name	"Beer"
	description	"Tactical Nuclear Penguin"
["name"], ["Pretzel"]	name	"Pretzel"
	description	"Glutino Pretzel Sticks"

Table 10.34. Content of the `associations_BasketContent` cache

KEY	ROW KEY	MAP ENTRIES	
["Basket_id"], ["davide_basket"]	["Basket_id", "products_name"], ["davide_basket", "Beer"]	Basket_id	"davide_basket"
		products_name	"Beer"
	["Basket_id", "products_name"], ["davide_basket", "Pretzel"]	Basket_id	"davide_basket"
		products_name	"Pretzel"

Example 10.17. Unidirectional one-to-many using maps with defaults

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

Table 10.35. Content of the `User` cache

KEY	MAP ENTRIES	
["id"], ["user_001"]	id	"user_001"

Table 10.36. Content of the `Address` cache

KEY	MAP ENTRIES	
["id"], ["address_001"]	id	"address_001"
	city	"Rome"
["id"], ["address_002"]	id	"address_002"
	city	"Paris"

Table 10.37. Content of the `associations_User_address` cache

KEY	ROW KEY	MAP ENTRIES	
["User_id"], "user_001"]	["User_id", "addresses_KEY"], ["user_001", "home"]	User_id	"user_001"
		addresses_KEY	"home"
		addresses_id	"address_001"
	["User_id", "addresses_KEY"], ["user_001", "work"]	User_id	"user_002"
		addresses_KEY	"work"
		addresses_id	"address_002"

Example 10.18. Unidirectional one-to-many using maps with @MapKeyColumn

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    @MapKeyColumn(name = "addressType")
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;
    // getters, setters ...
}
```

Table 10.38. Content of the `User` cache

KEY	MAP ENTRIES	
["id"], ["user_001"]	id	"user_001"

Table 10.39. Content of the `Address` cache

KEY	MAP ENTRIES	
["id"], ["address_001"]	id	"address_001"
	city	"Rome"
["id"], ["address_002"]	id	"address_002"
	city	"Paris"

Table 10.40. Content of the `associations_User_address` cache

KEY	ROW KEY	MAP ENTRIES	
["User_id"], "user_001"]	["User_id", "addressType"], ["user_001", "home"]	User_id	"user_001"
		addressesType	"home"
		addresses_id	"address_001"
	["User_id", "addressType"], ["user_001", "work"]	User_id	"user_002"
		addressesType	"work"
		addresses_id	"address_002"

Example 10.19. Unidirectional many-to-one

```
@Entity
public class JavaUserGroup {

    @Id
    private String jugId;
    private String name;

    // getters, setters ...
}

@Entity
public class Member {

    @Id
    private String id;
    private String name;
    @ManyToOne
    private JavaUserGroup memberOf;

    // getters, setters ...
}
```

Table 10.41. Content of the `JavaUserGroup` cache

KEY	MAP ENTRIES	
["jugId"], ["summer_camp"]	jugId	"summer_camp"
	name	"JUG Summer Camp"

Table 10.42. Content of the `Member` cache

KEY	MAP ENTRIES	
["member_id"], ["emmanuel"]	member_id	"emmanuel"
	name	"Emmanuel Bernard"
	memberOf_jug_id	"summer_camp"
["member_id"], ["jerome"]	member_id	"jerome"
	name	"Jerome"
	memberOf_jug_id	"summer_camp"

Example 10.20. Bidirectional many-to-one

```
@Entity
public class SalesForce {

    @Id
    private String id;
    private String corporation;

    @OneToMany(mappedBy = "salesForce")
    private Set<SalesGuy> salesGuys = new HashSet<SalesGuy>();

    // getters, setters ...
}

@Entity
public class SalesGuy {
    private String id;
    private String name;

    @ManyToOne
    private SalesForce salesForce;

    // getters, setters ...
}
```

Table 10.43. Content of the `SalesForce` cache

KEY	MAP ENTRIES	
["id"], ["red_hat"]	id	"red_hat"
	corporation	"Red Hat"

Table 10.44. Content of the `SalesGuy` cache

KEY	MAP ENTRIES	
["id"], ["eric"]	id	"eric"
	name	"Eric"
	salesForce_id	"red_hat"
["id"], ["simon"]	id	"simon"
	name	"Simon"
	salesForce_id	"red_hat"

Table 10.45. Content of the `associations_SalesGuy` cache

KEY	ROW KEY	MAP ENTRIES	
["salesForce_id"], ["red_hat"]	["salesForce_id", "id"], ["red_hat", "eric"]	salesForce_id	"red_hat"
		id	"eric"
	["salesForce_id", "id"], ["red_hat", "simon"]	salesForce_id	"red_hat"
		id	"simon"

Example 10.21. Unidirectional many-to-many

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();
    // getters, setters ...
}
```

The "Math" class has 2 students: John Doe and Mario Rossi

The "English" class has 2 students: Kate Doe and Mario Rossi

Table 10.46. Content of the `ClassRoom` cache

KEY	MAP ENTRIES	
["id"], [1]	id	1
	name	"Math"
["id"], [2]	id	2
	name	"English"

Table 10.47. Content of the `Student` cache

KEY	MAP ENTRIES	
["id"], ["john"]	id	"john"
	name	"John Doe"
["id"], ["mario"]	id	"mario"
	name	"Mario Rossi"
["id"], ["kate"]	id	"kate"
	name	"Kate Doe"

Table 10.48. Content of the `associations_ClassRoom_Student` cache

KEY	ROW KEY	MAP ENTRIES	
["ClassRoom_id"], [1]	["ClassRoom_id", "students_id"], [1, "mario"]	ClassRoom_id	1
		students_id	"mario"
	["ClassRoom_id", "students_id"], [1, "john"]	ClassRoom_id	1
		students_id	"john"
["ClassRoom_id"], [2]	["ClassRoom_id", "students_id"], [2, "kate"]	ClassRoom_id	2
		students_id	"kate"
	["ClassRoom_id", "students_id"], [2, "mario"]	ClassRoom_id	2
		students_id	"mario"

Example 10.22. Bidirectional many-to-many

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany( mappedBy = "bankAccounts" )
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

David owns 2 accounts: "012345" and "ZZZ-009"

Table 10.49. Content of the `AccountOwner` cache

KEY	MAP ENTRIES	
["id"], ["David"]	id	"David"
	SSN	"0123456"

Table 10.50. Content of the `BankAccount` cache

KEY	MAP ENTRIES	
["id"], ["account_1"]	id	"account_1"
	accountNumber	"X2345000"
["id"], ["account_2"]	id	"account_2"
	accountNumber	"ZZZ-009"

Table 10.51. Content of the `AccountOwner_BankAccount` cache

KEY	ROW KEY	MAP ENTRIES	
["bankAccounts_id"], ["account_1"]	["bankAccounts_id", "owners_id"], ["account_1", "David"]	bankAccounts_id	"account_1"
		owners_id	"David"
["bankAccounts_id"], ["account_2"]	["bankAccounts_id", "owners_id"], ["account_2", "David"]	bankAccounts_id	"account_2"
		owners_id	"David"
["owners_id"], ["David"]	["owners_id", "banksAccounts_id"], ["David", "account_1"]	bankAccounts_id	"account_1"
		owners_id	"David"
	["owners_id", "banksAccounts_id"], ["David", "account_2"]	bankAccounts_id	"account_2"
		owners_id	"David"

10.3. Transactions

While Ehcache technically supports transactions, Hibernate OGM is currently unable to use them. Careful!

If you need this feature, it should be easy to implement: contributions welcome! See *JIRA OGM-243* [https://hibernate.onjira.com/browse/OGM-243].

MongoDB

MongoDB [http://www.mongodb.org] is a document oriented datastore written in C++ with strong emphasis on ease of use. The nested nature of documents make it a particularly natural fit for most object representations.

This implementation is based upon the MongoDB Java driver. The currently supported version is 2.12.4.

11.1. Configuring MongoDB

Configuring Hibernate OGM to use MongoDb is easy:

- Add the MongoDB module and driver to the classpath
- provide the MongoDB URL to Hibernate OGM

11.1.1. Adding MongoDB dependencies

To add the dependencies via Maven, add the following module:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-mongodb</artifactId>
    <version>4.1.3.Final</version>
</dependency>
```

This will pull the MongoDB driver transparently.

If you’re not using a dependency management tool, copy all the dependencies from the distribution in the directories:

- `/lib/required`
- `/lib/mongodb`
- Optionally - depending on your container - you might need some of the jars from `/lib/provided`

11.1.2. MongoDB specific configuration properties

To get started quickly, pay attention to the following options:

- `hibernate.ogm.datastore.provider`
- `hibernate.ogm.datastore.host`

- `hibernate.ogm.datastore.database`

And we should have you running. The following properties are available to configure MongoDB support:

MongoDB datastore configuration properties

hibernate.ogm.datastore.provider
: To use MongoDB as a datastore provider, this property must be set to `mongodb`

hibernate.ogm.option.configurator
: The fully-qualified class name or an instance of a programmatic option configurator (see *Section 11.1.4, “Programmatic configuration”*)

hibernate.ogm.datastore.host
: The hostname of the MongoDB instance. The default value is `127.0.0.1`.

hibernate.ogm.datastore.port
: The port used by the MongoDB instance. The default value is `27017`.

hibernate.ogm.datastore.database
: The database to connect to. This property has no default value.

hibernate.ogm.datastore.create_database
: If set to true, the database will be created if it doesn’t exist. This property default value is false.

hibernate.ogm.datastore.username
: The username used when connecting to the MongoDB server. This property has no default value.

hibernate.ogm.datastore.password
: The password used to connect to the MongoDB server. This property has no default value. This property is ignored if the username isn’t specified.

hibernate.ogm.mongodb.connection_timeout
: Defines the timeout used by the driver when the connection to the MongoDB instance is initiated. This configuration is expressed in milliseconds. The default value is `5000`.

hibernate.ogm.mongodb.authentication_mechanism
: Define the authentication mechanism to use. Possible values are:

  - `GSSAPI`: The GSSAPI mechanism. See the *RFC* [http://tools.ietf.org/html/rfc4752]
  - `MONGODB_CR`: The MongoDB Challenge Response mechanism. This is the default value.
  - `MONGODB_X509`: The MongoDB X.509
  - `PLAIN`: The PLAIN mechanism. See the *RFC* [http://www.ietf.org/rfc/rfc4616.txt]

hibernate.ogm.datastore.document.association_storage
: Defines the way OGM stores association information in MongoDB. The following two strategies exist (values of the `org.hibernate.ogm.datastore.document.options.AssociationStorageType` enum):

  - `IN_ENTITY`: store association information within the entity
  - `ASSOCIATION_DOCUMENT`: store association information in a dedicated document per association

  `IN_ENTITY` is the default and recommended option unless the association navigation data is much bigger than the core of the document and leads to performance degradation.

hibernate.ogm.mongodb.association_document_storage
: Defines how to store assocation documents (applies only if the `ASSOCIATION_DOCUMENT` association storage strategy is used). Possible strategies are (values of the `org.hibernate.ogm.datastore.mongodb.options.AssociationDocumentStorageType` enum):

  - `GLOBAL_COLLECTION` (default): stores the association information in a unique MongoDB collection for all associations
  - `COLLECTION_PER_ASSOCIATION` stores the association in a dedicated MongoDB collection per association

hibernate.ogm.mongodb.write_concern
: Defines the write concern setting to be applied when issuing writes against the MongoDB datastore. Possible settings are (values of the `WriteConcernType` enum): `ERRORS_IGNORED`, `ACKNOWLEDGED`, `UNACKNOWLEDGED`, `FSYNCED`, `JOURNALED`, `REPLICA_ACKNOWLEDGED`, `MAJORITY` and `CUSTOM`. When set to `CUSTOM`, a custom `WriteConcern` implementation type has to be specified.

  This option is case insensitive and the default value is `ACKNOWLEDGED`.

hibernate.ogm.mongodb.write_concern_type
: Specifies a custom `WriteConcern` implementation type (fully-qualified name, class object or instance). This is useful in cases where the pre-defined configurations are not sufficient, e.g. if you want to ensure that writes are propagated to a specific number of replicas or given "tag set". Only takes effect if `hibernate.ogm.mongodb.write_concern` is set to `CUSTOM`.

hibernate.ogm.mongodb.read_preference
: Specifies the `ReadPreference` to be applied when issuing reads against the MongoDB datastore. Possible settings are (values of the `ReadPreferenceType` enum): `PRIMARY`, `PRIMARY_PREFERRED`, `SECONDARY`, `SECONDARY_PREFERRED` and `NEAREST`. It's currently not possible to plug in custom read preference types. If you're interested in such a feature, please let us know.

For more information, please refer to the *official documentation* [http://api.mongodb.org/java/current/com/mongodb/WriteConcern.html].

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `MongoDBProperties` when specifying the configuration properties listed above.

Common properties shared between stores are declared on `OgmProperties` (a super interface of `MongoDBProperties`).

For maximum portability between stores, use the most generic interface possible.

11.1.3. Annotation based configuration

Hibernate OGM allows to configure store-specific options via Java annotations. You can override global configurations for a specific entity or even a specify property by virtue of the location where you place that annotation.

When working with the MongoDB backend, you can specify the following settings:

- the write concern for entities and associations using the `@WriteConcern` annotation
- the read preference for entities and associations using the `@ReadPreference` annotation
- a strategy for storing associations using the `@AssociationStorage` and `@AssociationDocumentStorage` annotations (refer to *Section 11.2, "Storage principles"* to learn more about these options).

The following shows an example:

Example 11.1. Configuring the association storage strategy using annotations

```java
@Entity
@WriteConcern(WriteConcernType.JOURNALED)
@ReadPreference(ReadPreferenceType.PRIMARY_PREFERRED)
@AssociationStorage(AssociationStorageType.ASSOCIATION_DOCUMENT)
@AssociationDocumentStorage(AssociationDocumentStorageType.COLLECTION_PER_ASSOCIATION)
public class Zoo {

    @OneToMany
    private Set<Animal> animals;

    @OneToMany
    private Set<Person> employees;

    @OneToMany
    @AssociationStorage(AssociationStorageType.IN_ENTITY)
    private Set<Person> visitors;
    // getters, setters ...
}
```

The `@WriteConcern` annotation on the entity level expresses that all writes should be done using the `JOURNALED` setting. Similarly, the `@ReadPreference` annotation advices the engine to preferably read that entity from the primary node if possible. The other two annotations on the type-level specify that all associations of the `Zoo` class should be stored in separate assocation documents, using a dedicated collection per association. This setting applies to the `animals` and `employees` associations. Only the elements of the `visitors` association will be stored in the document of the corresponding `Zoo` entity as per the configuration of that specific property which takes precedence over the entity-level configuration.

11.1.4. Programmatic configuration

In addition to the annotation mechanism, Hibernate OGM also provides a programmatic API for applying store-specific configuration options. This can be useful if you can't modify certain entity types or don't want to add store-specific configuration annotations to them. The API allows set options in a type-safe fashion on the global, entity and property levels.

When working with MongoDB, you can currently configure the following options using the API:

- write concern
- read preference
- association storage strategy
- association document storage strategy

To set these options via the API, you need to create an `OptionConfigurator` implementation as shown in the following example:

Example 11.2. Example of an option configurator

```
public class MyOptionConfigurator extends OptionConfigurator {

    @Override
    public void configure(Configurable configurable) {
        configurable.configureOptionsFor( MongoDB.class )
            .writeConcern( WriteConcernType.REPLICA_ACKNOWLEDGED )
            .readPreference( ReadPreferenceType.NEAREST )
            .entity( Zoo.class )
                .associationStorage( AssociationStorageType.ASSOCIATION_DOCUMENT )
                .associationDocumentStorage( AssociationDocumentStorageType.COLLECTION_PER_ASSOCIATION )
                .property( "animals", ElementType.FIELD )
                    .associationStorage( AssociationStorageType.IN_ENTITY )
            .entity( Animal.class )
                .writeConcern( new RequiringReplicaCountOf( 3 ) )
                .associationStorage( AssociationStorageType.ASSOCIATION_DOCUMENT );
    }
}
```

The call to `configureOptionsFor()`, passing the store-specific identifier type `MongoDB`, provides the entry point into the API. Following the fluent API pattern, you then can configure global options (`writeConcern()`, `readPreference()`) and navigate to single entities or properties to apply options specific to these (`associationStorage()` etc.). The call to `writeConcern()` for the `Animal` entity shows how a specific write concern type can be used. Here `RequiringReplicaCountOf` is a custom implementation of `WriteConcern` which ensures that writes are propagated to a given number of replicas before a write is acknowledged.

Options given on the property level precede entity-level options. So e.g. the `animals` association of the `Zoo` class would be stored using the in entity strategy, while all other associations of the `Zoo` entity would be stored using separate association documents.

Similarly, entity-level options take precedence over options given on the global level. Global-level options specified via the API complement the settings given via configuration properties. In case a setting is given via a configuration property and the API at the same time, the latter takes precedence.

Note that for a given level (property, entity, global), an option set via annotations is overridden by the same option set programmatically. This allows you to change settings in a more flexible way if required.

To register an option configurator, specify its class name using the `hibernate.ogm.option.configurator` property. When bootstrapping a session factory or entity manager factory programmatically, you also can pass in an `OptionConfigurator` instance or the class object representing the configurator type.

11.2. Storage principles

Hibernate OGM tries to make the mapping to the underlying datastore as natural as possible so that third party applications not using Hibernate OGM can still read and update the same datastore. We worked particularly hard on the MongoDB model to offer various classic mappings between your object model and the MongoDB documents.

To describe things simply, each entity is stored as a MongoDB document. This document is stored in a MongoDB collection named after the entity type. The navigational information for each association from one entity to (a set of) entity is stored in the document representing the entity we are departing from.

11.2.1. Properties and built-in types

Each entity is represented by a document. Each property or more precisely column is represented by a field in this document, the field name being the column name.

Hibernate OGM supports by default the following property types:

- `java.lang.String`

```
{ "text" : "Hello world!" }
```

- `java.lang.Character` (or char primitive)

```
{ "delimiter" : "/" }
```

- `java.lang.Boolean` (or boolean primitive)

```
{ "favorite" : true }
```

- `java.lang.Byte` (or byte primitive)

```
{ "display_mask" : "70" }
```

- `java.lang.Byte[]` (or byte[])

```
{ "pdfAsBytes" : BinData(0,"MTIzNDU=") }
```

- `java.lang.Short` (or short primitive)

```
{ "urlPort" : 80 }
```

- `java.lang.Integer` (or integer primitive)

```
{ "stockCount" : 12309 }
```

- `java.lang.Long` (or long primitive)

```
{ "userId" : NumberLong("-6718902786625749549") }
```

- `java.lang.Float` (or float primitive)

```
{ "visitRatio" : 10.39 }
```

- `java.lang.Double` (or double primitive)

```
{ "tax_percentage" : 12.34 }
```

- `java.math.BigDecimal`

```
{ "site_weight" : "21.77" }
```

- `java.math.BigInteger`

```
{ "site_weight" : "444" }
```

- `java.util.Calendar`

```
{ "creation" : "2014/11/03 16:19:49:283 +0000" }
```

- `java.util.Date`

```
{ "last_update" : ISODate("2014-11-03T16:19:49.283Z") }
```

- `java.util.UUID`

```
{ "serialNumber" : "71f5713d-69c4-4b62-ad15-aed8ce8d10e0" }
```

- `java.util.URL`

```
{ "url" : "http://www.hibernate.org/" }
```

- `org.bson.types.ObjectId`

```
{ "object_id" : ObjectId("547d9b40e62048750f25ef77") }
```

Note

Hibernate OGM doesn’t store null values in MongoDB, setting a value to null is the same as removing the field in the corresponding object in the db.

This can have consequences when it comes to queries on null value.

11.2.2. Entities

Entities are stored as MongoDB documents and not as BLOBs: each entity property will be translated into a document field. You can use `@Table` and `@Column` annotations to rename respectively the collection the document is stored in and the document's field a property is persisted in.

Example 11.3. Default JPA mapping for an entity

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    // getters, setters ...
}
```

```
// Stored in the Collection "News"
{
    "_id" : "1234-5678-0123-4567",
    "title": "On the merits of NoSQL",
}
```

Example 11.4. Rename field and collection using @Table and @Column

```
@Entity
// Overrides the collection name
@Table(name = "News_Collection")
public class News {

    @Id
    private String id;

    // Overrides the field name
    @Column(name = "headline")
    private String title;

    // getters, setters ...
}
```

```
// Stored in the Collection "News"
{
    "_id" : "1234-5678-0123-4567",
    "headline": "On the merits of NoSQL",
}
```

11.2.2.1. Identifiers

> **Note**
>
> Hibernate OGM always store identifiers using the `_id` field of a MongoDB document ignoring the name of the property in the entity.
>
> That's a good thing as MongoDB has special treatment and expectation of the property `_id`.

An identifier type may be one of the *built-in types* or a more complex type represented by an embedded class. When you use a built-in type, the identifier is mapped like a regular property. When you use an embedded class, then the `_id` is representing a nested document containing the embedded class properties.

Example 11.5. Define an identifier as a primitive type

```
@Entity
public class Bookmark {

    @Id
    private String id;

    private String title;

    // getters, setters ...
}
```

```
{
  "_id" : "bookmark_1"
  "title" : "Hibernate OGM documentation"
}
```

Example 11.6. Define an identifier using @EmbeddedId

```
@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}

@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;
    private String content;

    // getters, setters ...
}
```

News collection as JSON in MongoDB

```
{
  "_id" : {
      "author" : "Guillaume",
      "title" : "How to use Hibernate OGM ?"
  },
  "content" : "Simple, just like ORM but with a NoSQL database"
}
```

Generally, it is recommended though to work with MongoDB's object id data type. This will facilitate the integration with other applications expecting that common MongoDB id type. To do so, you have two options:

- Define your id property as `org.bson.types.ObjectId`
- Define your id property as `String` and annotate it with `@Type(type="objectid")`

In both cases the id will be stored as native `ObjectId` in the datastore.

Example 11.7. Define an id as ObjectId

```
@Entity
public class News {

    @Id
    private ObjectId id;

    private String title;

    // getters, setters ...
}
```

Example 11.8. Define an id of type String as ObjectId

```
@Entity
```

```
public class News {

    @Id
    @Type(type = "objectid")
    private String id;

    private String title;

    // getters, setters ...
}
```

11.2.2.2. Identifier generation strategies

You can assign id values yourself or let Hibernate OGM generate the value using the `@GeneratedValue` annotation.

There are 4 different strategies:

1. *IDENTITY* (suggested)
2. *TABLE*
3. *SEQUENCE*
4. *AUTO*

1) IDENTITY generation strategy

The preferable strategy, Hibernate OGM will create the identifier upon insertion. To apply this strategy the id must be one of the following:

- annotated with `@Type(type="objectid")`
- `org.bson.types.ObjectId`

like in the following examples:

Example 11.9. Define an id of type String as ObjectId

```
@Entity
public class News {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Type(type = "objectid")
    private String id;

    private String title;

    // getters, setters ...
}
```

```
{
    "_id" : ObjectId("5425448830048b67064d40b1"),
    "title" : "Exciting News"
}
```

Example 11.10. Define an id as ObjectId

```
@Entity
public class News {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private ObjectId id;

    private String title;

    // getters, setters ...
}
```

```
{
    "_id" : ObjectId("5425448830048b67064d40b1"),
    "title" : "Exciting News"
}
```

2) TABLE generation strategy

Example 11.11. Id generation strategy TABLE using default values

```
@Entity
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Long id;

    private String name;

    // getters, setters ...
}
```

GuitarPlayer collection

```
{
    "_id" : NumberLong(1),
    "name" : "Buck Cherry"
}
```

hibernate_sequences collection

```
{
    "_id" : "GuitarPlayer",
    "next_val" : 101
}
```

Example 11.12. Id generation strategy TABLE using a custom table

```
@Entity
public class GuitarPlayer {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "guitarGen")
    @TableGenerator(
        name = "guitarGen",
        table = "GuitarPlayerSequence",
        pkColumnValue = "guitarPlayer",
        valueColumnName = "nextGuitarPlayerId"
    )
    private long id;

    // getters, setters ...
}
```

GuitarPlayer collection

```
{
    "_id" : NumberLong(1),
    "name" : "Buck Cherry"
}
```

GuitarPlayerSequence collection

```
{
    "_id" : "guitarPlayer",
    "nextGuitarPlayerId" : 2
}
```

3) SEQUENCE generation strategy

Example 11.13. SEQUENCE id generation strategy using default values

```
@Entity
public class Song {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Long id;

  private String title;

  // getters, setters ...
}
```

Song collection

```
{
  "_id" : NumberLong(2),
  "title" : "Flower Duet"
}
```

hibernate_sequences collection

```
{ "_id" : "song_sequence_name", "next_val" : 21 }
```

Example 11.14. SEQUENCE id generation strategy using custom values

```
@Entity
public class Song {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "songSequenceGenerator")
  @SequenceGenerator(
      name = "songSequenceGenerator",
      sequenceName = "song_seq",
      initialValue = 2,
      allocationSize = 20
  )
  private Long id;

  private String title;

  // getters, setters ...
}
```

Song collection

```
{
  "_id" : NumberLong(2),
  "title" : "Flower Duet"
}
```

hibernate_sequences collection

```
{ "_id" : "song_seq", "next_val" : 42 }
```

4) AUTO generation strategy

Warning

Care must be taken when using the `GenerationType.AUTO` strategy. When the property `hibernate.id.new_generator_mappings` is set to `false` (default), it will map to the `IDENTITY` strategy. As described before, this requires your ids to be of type `ObjectId` or @Type(type = "objectid") String. If `hibernate.id.new_generator_mappings` is set to true, `AUTO` will be mapped to the `TABLE` strategy. This requires your id to be of a numeric type.

We recommend to not use `AUTO` but one of the explicit strategies (`IDENTITY` or `TABLE`) to avoid potential misconfigurations.

For more details you can check the issue *OGM-663* [https://hibernate.atlassian.net/browse/OGM-663].

If the property `hibernate.id.new_generator_mappings` is set to `false`, `AUTO` will behave as the `IDENTITY` strategy.

If the property `hibernate.id.new_generator_mappings` is set to `true`, `AUTO` will behave as the `SEQUENCE` strategy.

Example 11.15. AUTO id generation strategy using default values

```
@Entity
public class DistributedRevisionControl {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  private String name;

  // getters, setters ...
}
```

DistributedRevisionControl collection

```
{ "_id" : NumberLong(1), "name" : "Git" }
```

hibernate_sequences collection

```
{ "_id" : "hibernate_sequence", "next_val" : 2 }
```

Example 11.16. AUTO id generation strategy wih `hibernate.id.new_generator_mappings` set to false and ObjectId

```
@Entity
public class Comedian {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private ObjectId id;

  private String name;

  // getters, setters ...
}
```

Comedian collection

```
{ "_id" : ObjectId("5458b11693f4add0f90519c5"), "name" : "Louis C.K." }
```

Example 11.17. Entity with @EmbeddedId

```
@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;

    // getters, setters ...
}

@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}
```

Rendered as JSON in MongoDB

```
{
    "_id" :{
        "title": "How does Hibernate OGM MongoDB work?",
        "author": "Guillaume"
    }
}
```

11.2.2.3. Embedded objects and collections

Hibernate OGM stores elements annotated with `@Embedded` or `@ElementCollection` as nested documents of the owning entity.

Example 11.18. Embedded object

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    @Embedded
    private NewsPaper paper;

    // getters, setters ...
}

@Embeddable
public class NewsPaper {

    private String name;
    private String owner;

    // getters, setters ...
}
```

```
{
    "_id" : "1234-5678-0123-4567",
    "title": "On the merits of NoSQL",
    "paper": {
        "name": "NoSQL journal of prophecies",
        "owner": "Delphy"
    }
}
```

Example 11.19. @ElementCollection with primitive types

```
@Entity
public class AccountWithPhone {

    @Id
    private String id;

    @ElementCollection
    private List<String> mobileNumbers;

    // getters, setters ...
}
```

AccountWithPhone collection

```
{
    "_id" : "john_account",
    "mobileNumbers" : [ "+1-222-555-0222", "+1-202-555-0333" ]
}
```

Example 11.20. @ElementCollection with one attribute

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

```
{
    "_id" : "df153180-c6b3-4a4c-a7da-d5de47cf6f00",
    "grandChildren" : [ "Luke", "Leia" ]
}
```

The class `GrandChild` has only one attribute `name`, this means that Hibernate OGM doesn’t need to store the name of the attribute.

If the nested document has two or more fields, like in the following example, Hibernate OGM will store the name of the fields as well.

Example 11.21. @ElementCollection with @OrderColumn

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    @OrderColumn( name = "birth_order" )
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

```
{
    "_id" : "e3e1ed4e-c685-4c3f-9a67-a5aeec6ff3ba",
    "grandChildren" :
        [
            {
                "name" : "Luke",
                "birth_order" : 0
            },
            {
                "name" : "Leia",
                "birthorder" : 1
            }
        ]
}
```

Note

You can override the column name used for a property of an embedded object. But you need to know that the default column name is the concatenation of the embedding property, a `.` (dot) and the embedded property (recursively for several levels of embedded objects).

The MongoDB datastore treats dots specifically as it transforms them into nested documents. If you want to override one column name and still keep the nested structure, don't forget the dots.

That's a bit abstract, so let's use an example.

```
@Entity
class Order {
    @Id String number;
    User user;
    Address shipping;
    @AttributeOverrides({
       @AttributeOverride(name="name", column=@Column(name="delivery.provider"),
        @AttributeOverride(name="expectedDelaysInDays", column=@Column(name="delivery.delays")
    })
    DeliveryProvider deliveryProvider;
    CreditCardType cardType;
}

// default columns
@Embedded
class User {
    String firstname;
    String lastname;
}

// override one column
@Embeddable
public Address {
    String street;
    @Column(name="shipping.dest_city")
    String city;
}

// both columns overridden from the embedding side
@Embeddable
public DeliveryProvider {
    String name;
    Integer expectedDelaysInDays;
}

// do not use dots in the overriding
// and mix levels (bad form)
@Embedded
class CreditCardType {
    String merchant;
    @Column(name="network")
    String network;
}
```

```
{
    "_id": "123RF33",
    "user": {
        "firstname": "Emmanuel",
        "lastname": "Bernard"
    },
    "shipping": {
        "street": "1 av des Champs Elysées",
        "dest_city": "Paris"
    },
    "delivery": {
        "provider": "Santa Claus Inc.",
        "delays": "1"
    }
    "network": "VISA",
    "cardType: {
        "merchant": "Amazon"
    }
}
```

If you share the same embeddable in different places, you can use JPA's `@AttributeOverride` to override columns from the embedding side. This is the case of `DeliveryProvider` in our example.

If you omit the dot in one of the columns, this column will not be part of the nested document. This is demonstrated by the `CreditCardType`. We advise you against it. Like crossing streams, it is bad form. This approach might not be supported in the future.

11.2.3. Associations

Hibernate OGM MongoDB proposes three strategies to store navigation information for associations. The three possible strategies are:

- *IN_ENTITY* (default)
- *ASSOCIATION_DOCUMENT*, using a global collection for all associations
- *COLLECTION_PER_ASSOCIATION*, using a dedicated collection for each association

To switch between these strategies, use of the three approaches to options:

- annotate your entity with `@AssocationStorage` and `@AssociationDocumentStorage` annotations (see *Section 11.1.3, "Annotation based configuration"*),
- use the API for programmatic configuration (see *Section 11.1.4, "Programmatic configuration"*)
- or specify a default strategy via the `hibernate.ogm.datastore.document.association_storage` and `hibernate.ogm.mongodb.association_document_storage` configuration properties.

11.2.3.1. In Entity strategy

- **-to-one associations*
- **-to-many associations*

In this strategy, Hibernate OGM stores the id(s) of the associated entity(ies) into the entity document itself. This field stores the id value for to-one associations and an array of id values for to-many associations. An embedded id will be represented by a nested document. For indexed collections (i.e. `List` or `Map`), the index will be stored along the id.

Note

When using this strategy the annotations `@JoinTable` will be ignored because no collection is created for associations.

You can use `@JoinColumn` to change the name of the field that stores the foreign key (as an example, see *Example 11.23, “Unidirectional one-to-one with @JoinColumn”*).

11.2.3.2. To-one associations

Example 11.22. Unidirectional one-to-one

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    private Vehicule vehicule;

    // getters, setters ...
}
```

```
{
  "_id" : "V_01",
  "brand" : "Mercedes"
}
```

Wheel collection as JSON in MongoDB

```
{
  "_id" : "W001",
  "diameter" : 0,
  "vehicule_id" : "V_01"
}
```

Example 11.23. Unidirectional one-to-one with @JoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @JoinColumn( name = "part_of" )
    private Vehicule vehicule;

    // getters, setters ...
}
```

```
{
  "_id" : "V_01",
  "brand" : "Mercedes"
}
```

Wheel collection as JSON in MongoDB

```
{
  "_id" : "W001",
  "diameter" : 0,
  "part_of" : "V_01"
}
```

In a true one-to-one association, it is possible to share the same id between the two entities and therefore a foreign key is not required. You can see how to map this type of association in the following example:

Example 11.24. Unidirectional one-to-one with @MapsId and @PrimaryKeyJoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}

@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @PrimaryKeyJoinColumn
    @MapsId
    private Vehicule vehicule;

    // getters, setters ...
}
```

Vehicule collection as JSON in MongoDB

```
{
  "_id" : "V_01",
  "brand" : "Mercedes"
}
```

Wheel collection as JSON in MongoDB

```
{
  "_id" : "V_01",
  "diameter" : 0,
}
```

Example 11.25. Bidirectional one-to-one

```
@Entity
public class Husband {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Wife wife;

    // getters, setters ...
}

@Entity
public class Wife {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Husband husband;

    // getters, setters ...
}
```

Husband collection as JSON in MongoDB

```
{
  "_id" : "alex",
  "name" : "Alex",
  "wife" : "bea"
}
```

Wife collection as JSON in MongoDB

```
{
  "_id" : "bea",
  "name" : "Bea",
  "husband" : "alex"
}
```

Example 11.26. Unidirectional many-to-one

```
@Entity
public class JavaUserGroup {

    @Id
    private String jugId;
    private String name;

    // getters, setters ...
}

@Entity
```

```
public class Member {

    @Id
    private String id;
    private String name;

    @ManyToOne
    private JavaUserGroup memberOf;

    // getters, setters ...
}
```

JavaUserGroup collection as JSON in MongoDB

```
{
    "_id" : "summer_camp",
    "name" : "JUG Summer Camp"
}
```

Member collection as JSON in MongoDB

```
{
    "_id" : "jerome",
    "name" : "Jerome"
    "memberOf_jugId" : "summer_camp"
}
{
    "_id" : "emmanuel",
    "name" : "Emmanuel Bernard"
    "memberOf_jugId" : "summer_camp"
}
```

Example 11.27. Bidirectional many-to-one

```
@Entity
public class SalesForce {

    @Id
    private String id;
    private String corporation;

    @OneToMany(mappedBy = "salesForce")
    private Set<SalesGuy> salesGuys = new HashSet<SalesGuy>();

    // getters, setters ...
}

@Entity
public class SalesGuy {
    private String id;
    private String name;
    @ManyToOne
    private SalesForce salesForce;

    // getters, setters ...
}
```

SalesForce collection

```
{
    "_id" : "red_hat",
    "corporation" : "Red Hat",
    "salesGuys" : [ "eric", "simon" ]
}
```

SalesGuy collection

```
{
    "_id" : "eric",
    "name" : "Eric"
    "salesForce_id" : "red_hat",
}
{
    "_id" : "simon",
    "name" : "Simon",
    "salesForce_id" : "red_hat"
}
```

Example 11.28. Bidirectional many-to-one between entities with embedded ids

```
@Entity
public class Game {

    @EmbeddedId
    private GameId id;

    private String name;

    @ManyToOne
    private Court playedOn;

    // getters, setters ...
}


public class GameId implements Serializable {

    private String category;
    @Column(name = "id.gameSequenceNo")
    private int sequenceNo;

    // getters, setters ...
    // equals / hashCode
}

@Entity
public class Court {

    @EmbeddedId
    private CourtId id;

    private String name;

    @OneToMany(mappedBy = "playedOn")
    private Set<Game> games = new HashSet<Game>();

    // getters, setters ...
}

public class CourtId implements Serializable {

    private String countryCode;
    private int sequenceNo;

    // getters, setters ...
    // equals / hashCode
}
```

Court collection.

```
{
    "_id" : {
        "countryCode" : "DE",
        "sequenceNo" : 123
    },
    "name" : "Hamburg Court",
    "games" : [
        { "gameSequenceNo" : 457, "category" : "primary" },
        { "gameSequenceNo" : 456, "category" : "primary" }
    ]
}
```

Game collection.

```
{
    "_id" : {
        "category" : "primary",
        "gameSequenceNo" : 456
    },
    "name" : "The game",
    "playedOn_id" : {
        "countryCode" : "DE",
            "sequenceNo" : 123
        }
}
{
    "_id" : {
        "category" : "primary",
        "gameSequenceNo" : 457
    },
    "name" : "The other game",
    "playedOn_id" : {
        "countryCode" : "DE",
        "sequenceNo" : 123
    }
}
```

Here we see that the embedded id is represented as a nested document and directly referenced by the associations.

11.2.3.3. To-many associations

Example 11.29. Unidirectional one-to-many

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide",
  "products" : [ "Beer", "Pretzel" ]
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

Example 11.30. Unidirectional one-to-many with @OrderColumn

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide",
  "products" : [
    {
      "products_name" : "Pretzel",
      "products_ORDER" : 1
    },
    {
      "products_name" : "Beer",
      "products_ORDER" : 0
    }
  ]
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

A map can be used to represents an association, in this case Hibernate OGM will store the key of the map and the associated id.

Example 11.31. Unidirectional one-to-many using maps with defaults

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

User collection as JSON in MongoDB

```
{
  "_id" : "user_001",
  "addresses" : [
    {
      "addresses_KEY" : "work",
      "addresses_id" : "address_001"
    },
    {
      "addresses_KEY" : "home",
      "addresses_id" : "address_002"
    }
  ]
}
```

Address collection as JSON in MongoDB

```
{ "_id" : "address_001", "city" : "Rome" }
{ "_id" : "address_002", "city" : "Paris" }
```

You can use @MapKeyColumn to rename the column containing the key of the map.

Example 11.32. Unidirectional one-to-many using maps with @MapKeyColumn

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    @MapKeyColumn(name = "addressType")
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

User collection as JSON in MongoDB

```
{
  "_id" : "user_001",
  "addresses" : [
    {
      "addressType" : "work",
      "addresses_id" : "address_001"
    },
    {
      "addressType" : "home",
      "addresses_id" : "address_002"
    }
  ]
}
```

Address collection as JSON in MongoDB

```
{ "_id" : "address_001", "city" : "Rome" }
{ "_id" : "address_002", "city" : "Paris" }
```

Example 11.33. Unidirectional many-to-many using in entity strategy

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

Student collection

```
{
  "_id" : "john",
  "name" :"John Doe" }
{
  "_id" : "mario",
  "name" : "Mario Rossi"
}
{
  "_id" : "kate",
  "name" : "Kate Doe"
}
```

ClassRoom collection

```
{
  "_id" : NumberLong(1),
  "lesson" : "Math"
  "students" : [
     "mario",
     "john"
  ]
}
{
  "_id" : NumberLong(2),
  "lesson" : "English"
  "students" : [
     "mario",
     "kate"
  ]
}
```

Example 11.34. Bidirectional many-to-many

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany( mappedBy = "bankAccounts" )
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

AccountOwner collection

```
{
    "_id" : "owner_1",
    "SSN" : "0123456"
    "bankAccounts" : [ "account_1" ]
}
```

BankAccount collection

```
{
    "_id" : "account_1",
    "accountNumber" : "X2345000"
    "owners" : [ "owner_1", "owner2222" ]
}
```

Example 11.35. Ordered list with embedded id

```
@Entity
public class Race {
    @EmbeddedId
    private RaceId raceId;

    @OrderColumn(name = "ranking")
    @OneToMany @JoinTable(name = "Race_Runners")
    private List<Runner> runnersByArrival = new ArrayList<Runner>();

    // getters, setters ...
}

public class RaceId implements Serializable {
    private int federationSequence;
    private int federationDepartment;

    // getters, setters, equals, hashCode
}

@Entity
public class Runner {
    @EmbeddedId
    private RunnerId runnerId;
    private int age;

    // getters, setters ...
}

public class RunnerId implements Serializable {
    private String firstname;
    private String lastname;

    // getters, setters, equals, hashCode
}
```

Race collection.

```
{
    "_id": {
        "federationDepartment": 75,
        "federationSequence": 23
    },
    "runnersByArrival": [{
        "firstname": "Pere",
        "lastname": "Noel",
        "ranking": 1
    }, {
        "firstname": "Emmanuel",
        "lastname": "Bernard",
        "ranking": 0
    }]
}
```

Runner collection.

```
{
    "_id": {
        "firstname": "Pere",
        "lastname": "Noel"
    },
    "age": 105
} {
    "_id": {
        "firstname": "Emmanuel",
        "lastname": "Bernard"
    },
    "age": 37
}
```

11.2.3.4. One collection per association strategy

In this strategy, Hibernate OGM creates a MongoDB collection per association in which it will store all navigation information for that particular association.

This is the strategy closest to the relational model. If an entity A is related to B and C, 2 collections will be created. The name of this collection is made of the association table concatenated with `associations_`.

For example, if the `BankAccount` and `Owner` are related, the collection used to store will be named `associations_Owner_BankAccount`. You can rename The prefix is useful to quickly identify the association collections from the entity collections. You can also decide to rename the collection representing the association using `@JoinTable` (see *an example*)

Each document of an association collection has the following structure:

- `_id` contains the id of the owner of relationship
- `rows` contains all the id of the related entities

> **Note**
>
> The preferred approach is to use the *in-entity strategy* but this approach can alleviate the problem of having documents that are too big.

Example 11.36. Unidirectional relationship

```
{
    "_id" : { "owners_id" : "owner0001" },
    "rows" : [
        "accountABC",
        "accountXYZ"
    ]
}
```

Example 11.37. Bidirectional relationship

```
{
    "_id" : { "owners_id" : "owner0001" },
    "rows" : [ "accountABC", "accountXYZ" ]
}
{
    "_id" : { "bankAccounts_id" : "accountXYZ" },
    "rows" : [ "owner0001" ]
}
```

> **Note**
>
> This strategy won't affect *-to-one associations or embedded collections.

Example 11.38. Unidirectional one-to-many using one collection per strategy

```
@Entity
public class Basket {

    @Id
    private String id;
    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide"
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

associations_Basket_Product collection

```
{
  "_id" : { "Basket_id" : "davide_basket" },
  "rows" : [ "Beer", "Pretzel" ]
}
```

The order of the element in the list might be preserved using @OrderColumn. Hibernate OGM will store the order adding an additional fieldd to the document containing the association.

Example 11.39. Unidirectional one-to-many using one collection per strategy with @OrderColumn

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    @OrderColumn
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide"
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

associations_Basket_Product collection

```
{
  "_id" : { "Basket_id" : "davide_basket" },
  "rows" : [
    {
      "products_name" : "Pretzel",
      "products_ORDER" : 1
    },
    {
      "products_name" : "Beer",
      "products_ORDER" : 0
    }
  ]
}
```

Example 11.40. Unidirectional many-to-many using one collection per association strategy

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

Student collection

```
{
  "_id" : "john",
  "name" : "John Doe"
}
{
  "_id" : "mario",
  "name" : "Mario Rossi"
}
{
  "_id" : "kate",
  "name" : "Kate Doe"
}
```

ClassRoom collection

```
{
  "_id" : NumberLong(1),
  "lesson" : "Math"
}
{
  "_id" : NumberLong(2),
  "lesson" : "English"
}
```

associations_ClassRoom_Student

```
{
  "_id" : {
    "ClassRoom_id" : NumberLong(1),
  },
  "rows" : [ "john", "mario" ]
}
{
  "_id" : {
    "ClassRoom_id" : NumberLong(2),
  },
  "rows" : [ "mario", "kate" ]
}
```

Example 11.41. Bidirectional many-to-many using one collection per association strategy

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany(mappedBy = "bankAccounts")
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

AccountOwner collection

```
{
  "_id" : "owner_1",
  "SSN" : "0123456"
}
```

BankAccount collection

```
{
  "_id" : "account_1",
  "accountNumber" : "X2345000"
}
```

associations_AccountOwner_BankAccount collection

```
{
  "_id" : {
    "bankAccounts_id" : "account_1"
  },
  "rows" : [ "owner_1" ]
}
{
  "_id" : {
    "owners_id" : "owner_1"
  },
  "rows" : [ "account_1" ]
}
```

You can change the name of the collection containing the association using the `@JoinTable` annotation. In the following example, the name of the collection containing the association is `OwnerBankAccounts` (instead of the default `associations_AccountOwner_BankAccount`)

Example 11.42. Bidirectional many-to-many using one collection per association strategy and @JoinTable

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    @JoinTable( name = "OwnerBankAccounts" )
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany(mappedBy = "bankAccounts")
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

AccountOwner collection

```
{
  "_id" : "owner_1",
  "SSN" : "0123456"
}
```

BankAccount collection

```
{
  "_id" : "account_1",
  "accountNumber" : "X2345000"
}
```

OwnerBankAccount

```
{
  "_id" : {
    "bankAccounts_id" : "account_1"
  },
  "rows" : [ "owner_1" ]
}
{
  "_id" : {
    "owners_id" : "owner_1"
  },
  "rows" : [ "account_1" ]
}
```

11.2.3.5. Global collection strategy

With this strategy, Hibernate OGM creates a single collection named `Associations` in which it will store all navigation information for all associations. Each document of this collection is structured in 2 parts. The first is the `_id` field which contains the identifier information of the association owner and the name of the association table. The second part is the `rows` field which stores (into an embedded collection) all ids that the current instance is related to.

Note

This strategy won't affect *-to-one associations or embedded collections.

Generally, you should not make use of this strategy unless embedding the association information proves to be too big for your document and you wish to separate them.

Example 11.43. Associations collection containing unidirectional association

```
{
    "_id": {
        "owners_id": "owner0001",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [ "accountABC", "accountXYZ" ]
}
```

For a bidirectional relationship, another document is created where ids are reversed. Don't worry, Hibernate OGM takes care of keeping them in sync:

Example 11.44. Associations collection containing a bidirectional association

```
{
    "_id": {
        "owners_id": "owner0001",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [ "accountABC", "accountXYZ" ]
}
{
    "_id": {
        "bankAccounts_id": "accountXYZ",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [ "owner0001" ]
}
```

Example 11.45. Unidirectional one-to-many using global collection strategy

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide"
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

Associations collection

```
{
  "_id" : {
    "Basket_id" : "davide_basket",
    "table" : "Basket_Product"
  },
  "rows" : [
    {
      "products_name" : "Pretzel",
      "products_ORDER" : 1
    },
    {
      "products_name" : "Beer",
    "products_ORDER" : 0
    }
  ]
}
```

Example 11.46. Unidirectional one-to-many using global collection strategy with `@JoinTable`

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    // It will change the value stored in the field table in the Associations collection
    @JoinTable( name = "BasketContent" )
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;
    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide"
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

Associations collection

```
{
  "_id" : {
    "Basket_id" : "davide_basket",
    "table" : "BasketContent"
  },
  "rows" : [ "Beer", "Pretzel" ]
}
```

Example 11.47. Unidirectional many-to-many using global collection strategy

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {
    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

Student collection

```
{
  "_id" : "john",
  "name" : "John Doe"
}
{
  "_id" : "mario",
  "name" : "Mario Rossi"
}
{
  "_id" : "kate",
  "name" : "Kate Doe"
}
```

ClassRoom collection

```
{
  "_id" : NumberLong(1),
  "lesson" : "Math"
}
{
  "_id" : NumberLong(2),
  "lesson" : "English"
}
```

Associations collection

```
{
  "_id" : {
    "ClassRoom_id" : NumberLong(1),
    "table" : "ClassRoom_Student"
  },
  "rows" : [ "john", "mario" ]
}
{
  "_id" : {
    "ClassRoom_id" : NumberLong(2),
    "table" : "ClassRoom_Student"
  },
  "rows" : [ "mario", "kate" ]
}
```

Example 11.48. Bidirectional many-to-many using global collection strategy

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany(mappedBy = "bankAccounts")
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

AccountOwner collection

```
{
  "_id" : "owner0001",
  "SSN" : "0123456"
}
```

BankAccount collection

```
{
  "_id" : "account_1",
  "accountNumber" : "X2345000"
}
```

Associations collection

```
{
  "_id" : {
    "bankAccounts_id" : "account_1",
    "table" : "AccountOwner_BankAccount"
    },

  "rows" : [ "owner0001" ]
}
{
  "_id" : {
    "owners_id" : "owner0001",
    "table" : "AccountOwner_BankAccount"
  },

  "rows" : [ "account_1" ]
}
```

11.3. Transactions

MongoDB does not support transactions. Only changes applied to the same document are done atomically. A change applied to more than one document will not be applied atomically. This problem is slightly mitigated by the fact that Hibernate OGM queues all changes before applying them during flush time. So the window of time used to write to MongoDB is smaller than what you would have done manually.

We recommend that you still use transaction demarcations with Hibernate OGM to trigger the flush operation transparently (on commit). But do not consider rollback as a possibility, this won't work.

11.4. Optimistic Locking

MongoDB does not provide a built-in mechanism for detecting concurrent updates to the same document but it provides a way to execute atomic find and update operations. By exploiting this commands Hibernate OGM can detect concurrent modifications to the same document.

You can enable optimistic locking detection using the annotation `@Version`:

Example 11.49. Optimistic locking detection via `@Version`

```
@Entity
public class Planet implements Nameable {

    @Id
    private String id;
    private String name;

    @Version
    private int version;

   // getters, setters ...
}
```

```
{
  "_id" : "planet-1",
  "name" : "Pluto",
  "version" : 0
}
```

The `@Version` annotation define which attribute will keep track of the version of the document, Hibernate OGM will update the field when required and if two changes from two different sessions (for example) are applied to the same document a `org.hibernate.StaleObjectStateException` is thrown.

You can use `@Column` to change the name of the field created on MongoDB:

Example 11.50. Optimistic locking detection via `@Version` using `@Column`

```
@Entity
public class Planet implements Nameable {

    @Id
    private String id;
    private String name;

    @Version
    @Column(name="OPTLOCK")
    private int version;

   // getters, setters ...
}
```

```
{
  "_id" : "planet-1",
  "name" : "Pluto",
  "OPTLOCK" : 0
}
```

11.5. Queries

You can express queries in a few different ways:

- using JP-QL
- using a native MongoQL query
- using a Hibernate Search query (brings advanced full-text and geospatial queries)

While you can use JP-QL for simple queries, you might hit limitations. The current recommended approach is to use native MongoQL if your query involves nested (list of) elements.

Note

In order to reflect changes performed in the current session, all entities affected by a given query are flushed to the datastore prior to query execution (that's the case for Hibernate ORM as well as Hibernate OGM).

For not fully transactional stores such as MongoDB this can cause changes to be written as a side-effect of running queries which cannot be reverted by a possible later rollback.

Depending on your specific use cases and requirements you may prefer to disable auto-flushing, e.g. by invoking `query.setFlushMode( FlushMode.MANUAL )`. Bear in mind though that query results will then not reflect changes applied within the current session.

11.5.1. JP-QL queries

Hibernate OGM is a work in progress, so only a sub-set of JP-QL constructs is available when using the JP-QL query support. This includes:

- simple comparisons using "<", "#", "=", ">=" and ">"
- `IS NULL` and `IS NOT NULL`
- the boolean operators `AND`, `OR`, `NOT`
- `LIKE`, `IN` and `BETWEEN`
- `ORDER BY`

Queries using these constructs will be transformed into equivalent native MongoDB queries.

Note

Let us know *by opening an issue or sending an email* what query you wish to execute. Expanding our support in this area is high on our priority list.

11.5.2. Native MongoDB queries

Hibernate OGM also supports certain forms of native queries for MongoDB. Currently two forms of native queries are available via the MongoDB backend:

- find queries specifying the search criteria only
- queries specified using the MongoDB CLI syntax

The former always maps results to entity types. The latter either maps results to entity types or to certain supported forms of projection. Note that parameterized queries are not supported by MongoDB, so don't expect `Query#setParameter()` to work.

Warning

Specifying native MongoDB queries using the CLI syntax is an EXPERIMENTAL feature for the time being. Currently only `find()` and `count()` queries are supported via the CLI syntax. Further query types (including updating queries) may be supported in future revisions.

No cursor operations such as `sort()` are supported. Instead use the corresponding MongoDB *query modifiers* [http://docs.mongodb.org/manual/reference/operator/query-modifier/] such as `$orderby` within the criteria parameter.

JSON parameters passed via the CLI syntax must be specified using the *strict mode* [http://docs.mongodb.org/manual/reference/mongodb-extended-json/] The only relaxation of this is that single quotes may be used when specifying attribute names/values to facilitate embedding queries within Java strings.

Note that results of projections are returned as retrieved from the MongoDB driver at the moment and are not (yet) converted using suitable Hibernate OGM type implementations.

You can execute native queries as shown in the following example:

Example 11.51. Using the JPA API

```
@Entity
public class Poem {

    @Id
    private Long id;

    private String name;

    private String author;

   // getters, setters ...
}

...

javax.persistence.EntityManager em = ...

// criteria-only find syntax
String query1 = "{ $and: [ { name : 'Portia' }, { author : 'Oscar Wilde' } ] }";
Poem poem = (Poem) em.createNativeQuery( query1, Poem.class ).getSingleResult();
```

```
// criteria-only find syntax with order-by
String query2 = "{ $query : { author : 'Oscar Wilde' }, $orderby : { name : 1 } }";
List<Poem> poems = em.createNativeQuery( query2, Poem.class ).getResultList();

// projection via CLI-syntax
String query3 = "db.WILDE_POEM.find(" +
    "{ '$query' : { 'name' : 'Athanasia' }, '$orderby' : { 'name' : 1 } }" +
    "{ 'name' : 1 }" +
    ")";

// will contain name and id as MongoDB always returns the id for projections
List<Object[]> poemNames = (List<Object[]>)em.createNativeQuery( query3 ).getResultList();

// projection via CLI-syntax
String query4 = "db.WILDE_POEM.count({ 'name' : 'Athanasia' })";

Object[] count = (Object[])em.createNativeQuery( query4 ).getSingleResult();
```

The result of a query is a managed entity (or a list thereof) or a projection of attributes in form of an object array, just like you would get from a JP-QL query.

Example 11.52. Using the Hibernate native API

```
OgmSession session = ...

String query1 = "{ $and: [ { name : 'Portia' }, { author : 'Oscar Wilde' } ] }";
Poem poem = session.createNativeQuery( query1 )
                      .addEntity( "Poem", Poem.class )
                      .uniqueResult();

String query2 = "{ $query : { author : 'Oscar Wilde' }, $orderby : { name : 1 } }";
List<Poem> poems = session.createNativeQuery( query2 )
                      .addEntity( "Poem", Poem.class )
                      .list();
```

Native queries can also be created using the `@NamedNativeQuery` annotation:

Example 11.53. Using @NamedNativeQuery

```
@Entity
@NamedNativeQuery(
   name = "AthanasiaPoem",
   query = "{ $and: [ { name : 'Athanasia' }, { author : 'Oscar Wilde' } ] }",
   resultClass = Poem.class )
public class Poem { ... }

...

// Using the EntityManager
Poem poem1 = (Poem) em.createNamedQuery( "AthanasiaPoem" )
                     .getSingleResult();
```

```
// Using the Session
Poem poem2 = (Poem) session.getNamedQuery( "AthanasiaPoem" )
                     .uniqueResult();
```

Hibernate OGM stores data in a natural way so you can still execute queries using the MongoDB driver, the main drawback is that the results are going to be raw MongoDB documents and not managed entities.

11.5.3. Hibernate Search

You can index your entities using Hibernate Search. That way, a set of secondary indexes independent of MongoDB is maintained by Hibernate Search and you can write queries on top of them. The benefit of this approach is a nice integration at the JPA / Hibernate API level (managed entities are returned by the queries). The drawback is that you need to store the Lucene indexes somewhere (file system, infinispan grid, etc). Have a look at the Infinispan section (*Section 9.6, “Storing a Lucene index in Infinispan”*) for more info on how to use Hibernate Search.

Neo4j

Neo4j [http://www.neo4j.org] is a robust (fully ACID) transactional property graph database. This kind of databases are suited for those type of problems that can be represented with a graph like social relationships or road maps for example.

At the moment only the support for the embedded Neo4j is included in OGM.

12.1. How to add Neo4j integration

1. Add the dependencies to your project. If your project uses Maven you can add this to the pom.xml:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-neo4j</artifactId>
    <version>4.1.3.Final</version>
</dependency>
```

Alternatively you can find the required libraries in the distribution package on *SourceForge* [https://downloads.sourceforge.net/project/hibernate/hibernate-ogm/4.1.3.Final/hibernate-ogm-4.1.3.Final-dist.zip]

2. Add the following properties:

```
hibernate.ogm.datastore.provider = neo4j_embedded
hibernate.ogm.neo4j.database_path = C:\example\mydb
```

12.2. Configuring Neo4j

The following properties are available to configure Neo4j support:

Neo4j datastore configuration properties

hibernate.ogm.neo4j.database_path
: The absolute path representing the location of the Neo4j database. Example: `C:\neo4jdb\mydb`

hibernate.ogm.neo4j.configuration_resource_name (optional)
: Location of the Neo4j embedded properties file. It can be an URL, name of a classpath resource or file system path.

hibernate.schema_update.unique_constraint_strategy (optional)
: If set to `SKIP`, Hibernate OGM won't create any unique constraints on the nodes representing the entities. This property won't affect the unique

constraints generated for sequences. Other possible values (defined on the `org.hibernate.tool.hbm2ddl.UniqueConstraintSchemaUpdateStrategy` enum) are `DROP_RECREATE_QUIETLY` and `RECREATE_QUIETLY` but the effect will be the same (since Neo4j constraints don't have a name): keep the existing constraints and create the missing one. Default value is `DROP_RECREATE_QUIETLY`.

Warning

At the moment, you must not specify the property `hibernate.transaction.jta.platform` when using Neo4j. It would override Hibernate OGM's version and the transactions would not work correctly.

Note

Use the following method to retrieve the transaction manager (it needs to be done after the `EntityManagerFactory` has been bootstrapped):

```
private static TransactionManager extractJBossTransactionManager(EntityManagerFactory factory) {
    SessionFactoryImplementor sessionFactory =
        (SessionFactoryImplementor) ( (HibernateEntityManagerFactory) factory ).getSessionFactory()
    return sessionFactory.getServiceRegistry().getService( JtaPlatform.class ).retrieveTransactionM
}
```

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `Neo4jProperties` when specifying the configuration properties listed above.

Common properties shared between stores are declared on `OgmProperties` (a super interface of `Neo4jProperties`).

For maximum portability between stores, use the most generic interface possible.

12.3. Storage principles

Hibernate OGM tries to make the mapping to the underlying datastore as natural as possible so that third party applications not using Hibernate OGM can still read and update the same datastore.

To make things simple, each entity is represented by a node, each embedded object is also represented by a node. Links between entities (whether to-one to to-many associations) are represented by relationships between nodes. Entity and embedded nodes are labelled `ENTITY` and `EMBEDDED` respectively.

12.3.1. Properties and built-in types

Each entity is represented by a node. Each property or more precisely column is represented by an attribute of this node.

The following types (and corresponding primitives) get passed to Neo4j without any conversion:

- `java.lang.Boolean`
- `java.lang.Character`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.String`

The following types get converted into `java.lang.String`:

- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.util.Calendar`

  ```
  stored as +String+ with the format "yyyy/MM/dd HH:mm:ss:SSS Z"
  ```

- `java.util.Date`

  ```
  stored as +String+ with the format "yyyy/MM/dd HH:mm:ss:SSS Z"
  ```

- `java.util.UUID`
- `java.util.URL`

Note

Hibernate OGM doesn’t store null values in Neo4J, setting a value to null is the same as removing the corresponding entry from Neo4J.

This can have consequences when it comes to queries on null value.

12.3.2. Entities

Entities are stored as Neo4j nodes, which means each entity property will be translated into a property of the node. The name of the table mapping the entity is used as label.

You can use the name property of the `@Table` and `@Column` annotations to rename the label and the node’s properties.

An additional label `ENTITY` is added to the node.

Example 12.1. Default JPA mapping for an entity

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    // getters, setters ...
}
```

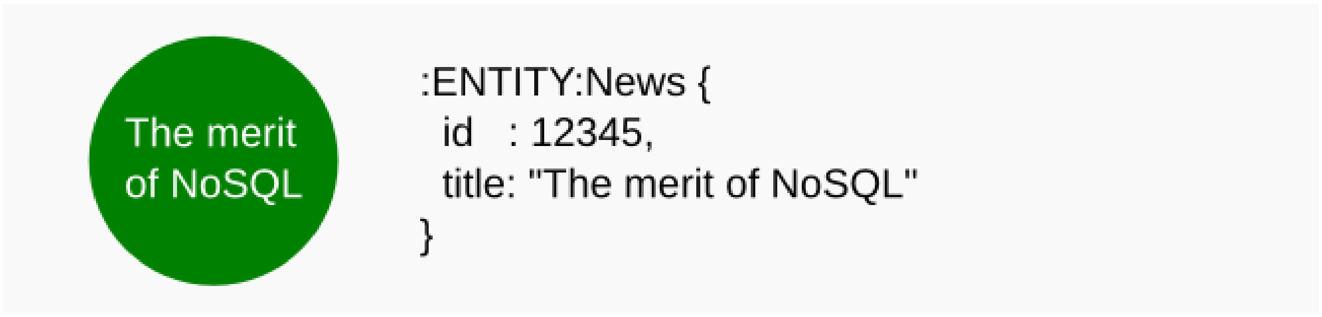


Figure 12.1.

Example 12.2. Rename node label and properties using @Table and @Column

```
@Entity
@Table(name="ARTICLE")
public class News {

    @Id
    private String id;

    @Column(name = "headline")
    private String title;

    // getters, setters ...
}
```

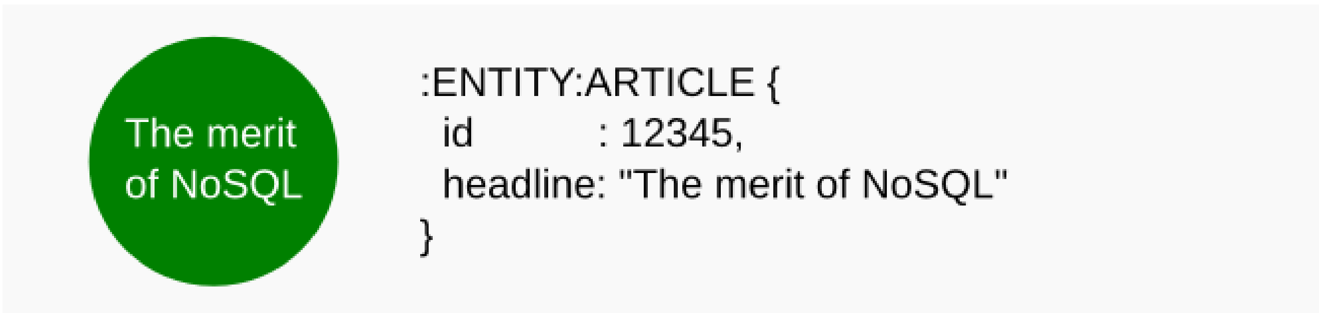


Figure 12.2.

12.3.2.1. Identifiers and unique constraints

Warning

Neo4j does not support constraints on more than one property. For this reason, Hibernate OGM will create a unique constraint ONLY when it spans a single property and it will ignore the ones spanning multiple properties.

The lack of unique constraints on node properties might result in the creation of multiple nodes with the same identifier.

Hibernate OGM will create unique constraints for the identifier of entities and for the properties annotated with:

- `@Id`
- `@EmbeddedId`
- `@NaturalId`
- `@Column( unique = true )`
- `@Table( uniqueConstraints = @UniqueConstraint(columnNames = { "column_name" } ) )`

Embedded identifiers are currently stored as dot separated properties.

Example 12.3. Entity with @EmbeddedId

```
@Entity
```

```
public class News {

    @EmbeddedId
    private NewsID newsId;

    private String content

    // getters, setters ...
}

@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}
```

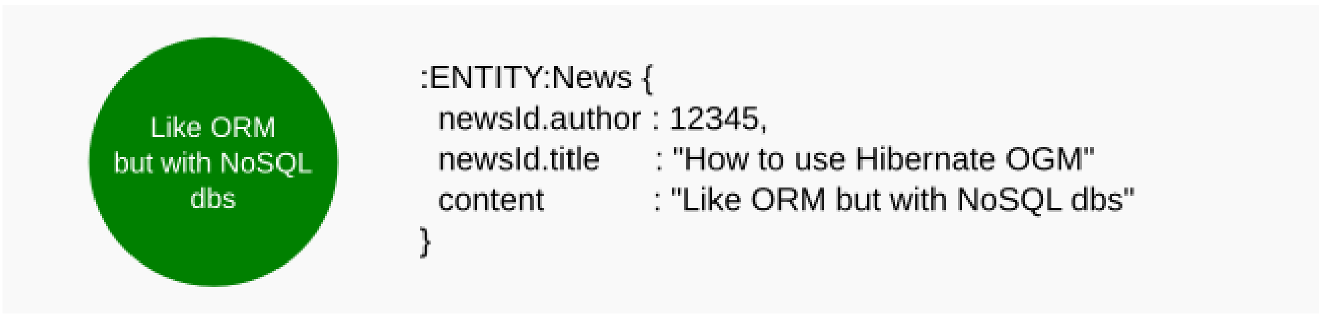

Like ORM but with NoSQL dbs
:ENTITY:News {
newsId.author : 12345,
newsId.title : "How to use Hibernate OGM"
content : "Like ORM but with NoSQL dbs"
}

Figure 12.3.

12.3.2.2. Embedded objects and collections

Embedded elements are stored as separate nodes labeled with `EMBEDDED`.

The the type of the relationship that connects the entity node to the embedded node is the attribute name representing the embedded in the java class.

Example 12.4. Embedded object

```
@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;

    @Embedded
    private NewsPaper paper;

    // getters, setters ...
}
```

```
@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}

@Embeddable
public class NewsPaper {

    private String name;
    private String owner;

    // getters, setters ...
}
```

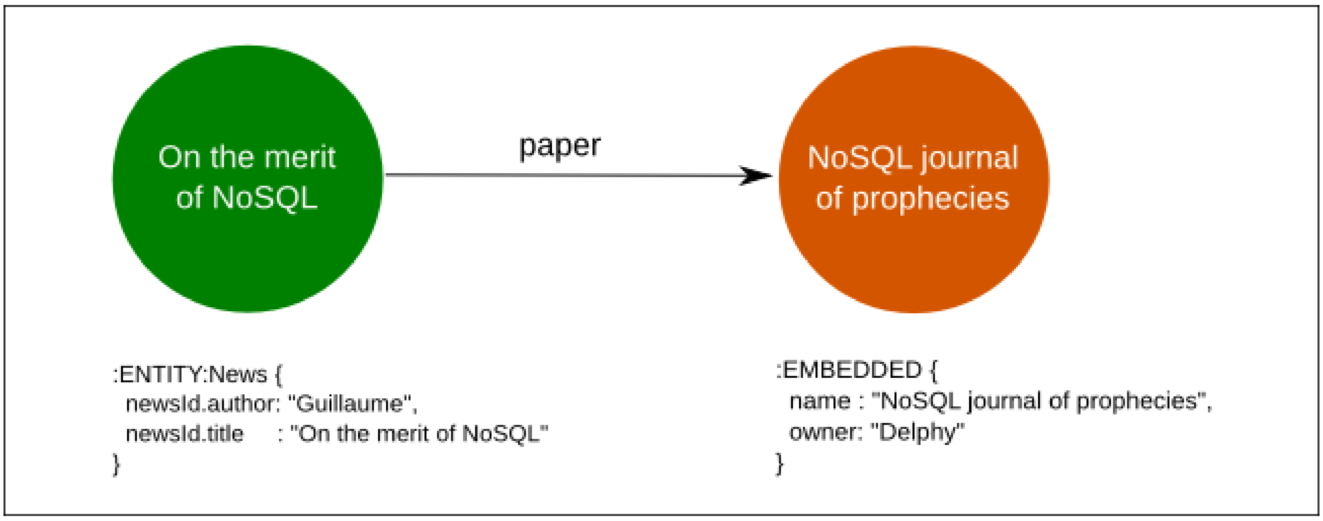

On the merit of NoSQL
paper
NoSQL journal of prophecies
:ENTITY:News {
newsId.author: "Guillaume",
newsId.title : "On the merit of NoSQL"
}
:EMBEDDED {
name : "NoSQL journal of prophecies",
owner: "Delphy"
}

Figure 12.4.

Example 12.5. @ElementCollection

```

@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {
    private String name;

    // getters, setters ...
}
```

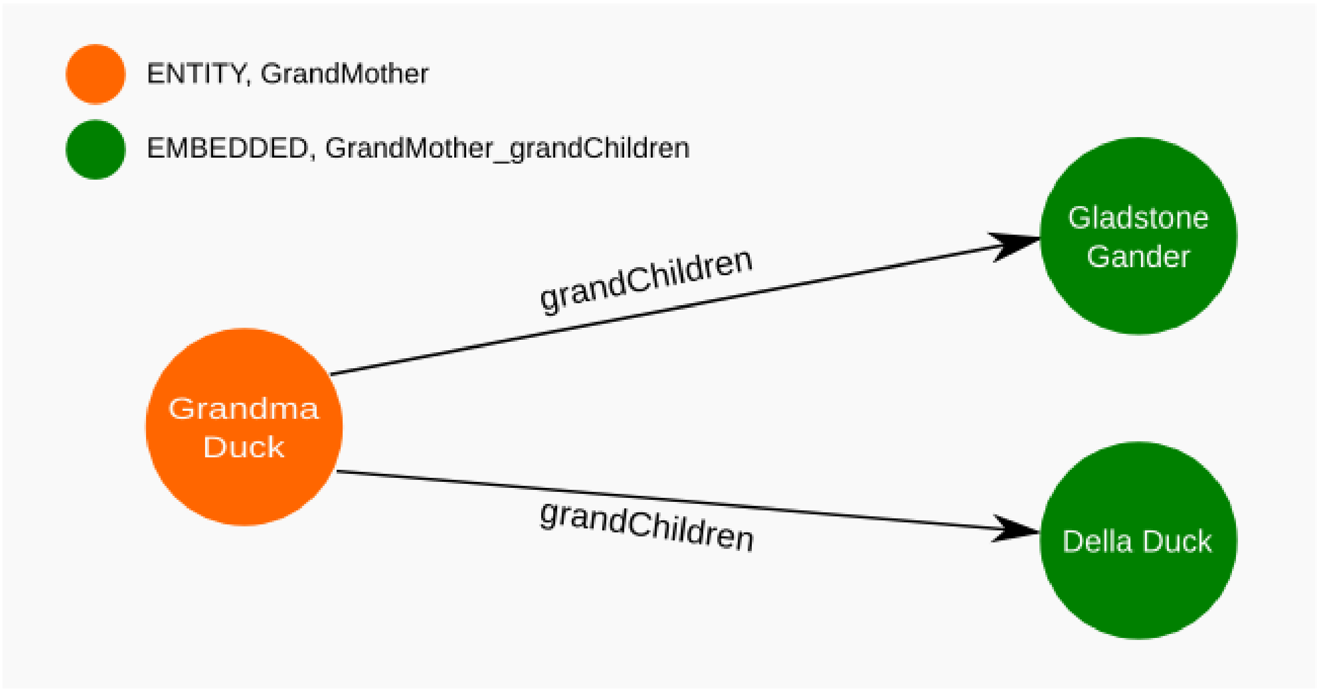


Figure 12.5.

Note that in the previous examples no property is added to the relationships; in the following one, one property is added to keep track of the order of the elements in the list.

Example 12.6. @ElementCollection with @OrderColumn

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    @OrderColumn( name = "birth_order" )
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

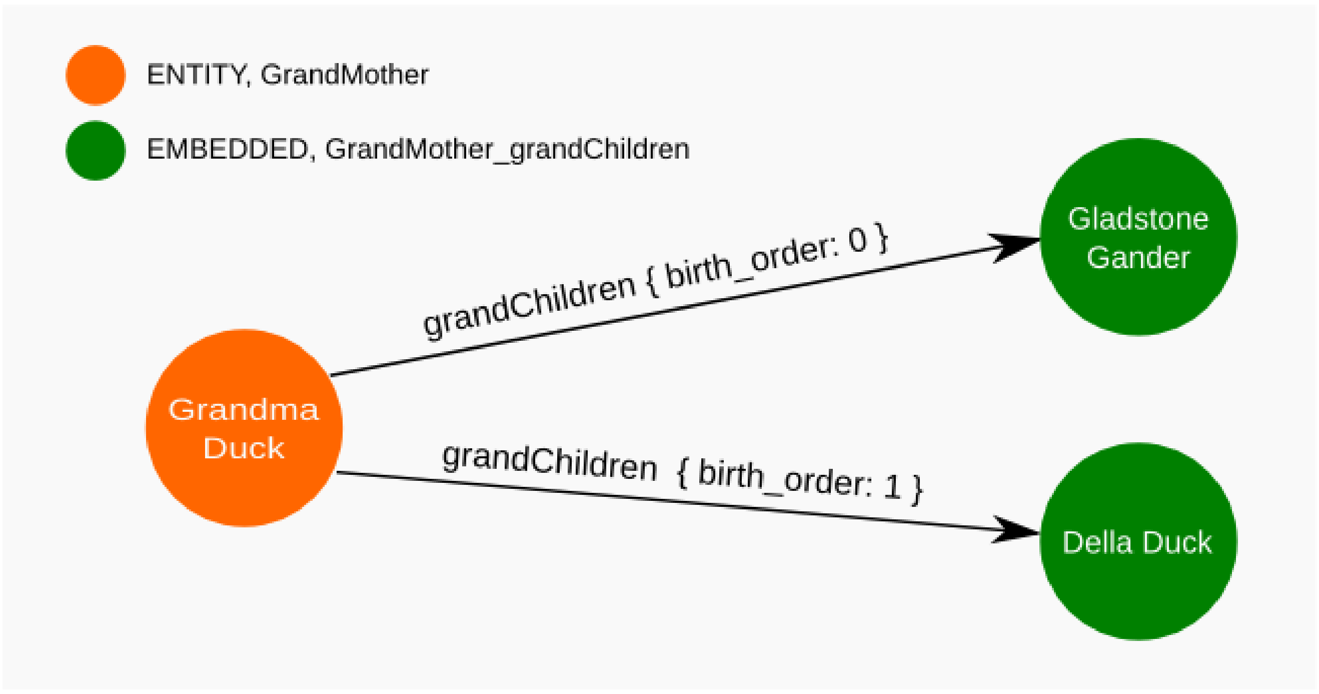

ENTITY, GrandMother
EMBEDDED, GrandMother_grandChildren
Grandma Duck
Gladstone Gander
Della Duck
grandChildren { birth_order: 0 }
grandChildren { birth_order: 1 }

Figure 12.6.

12.3.3. Associations

An association, bidirectional or unidirectional, is always mapped using one relationship, beginning at the owning side of the association. This is possible because in Neo4j relationships can be navigated in both directions.

The type of the relationships depends on the type of the association, but in general it is the role of the association on the main side. The only property stored on the relationship is going to be the index of the association when required, for example when the association is annotated with `@OrderColumn` or when a `java.util.Map` is used.

In Neo4j nodes are connected via relationship, this means that we don’t need to create properties which store foreign column keys. This means that annotation like `@JoinColumn` won’t have any effect.

Example 12.7. Unidirectional one-to-one

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;
    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private String company;
    private double diameter;

    @OneToOne
    private Vehicule vehicule;

    // getters, setters ...
}
```

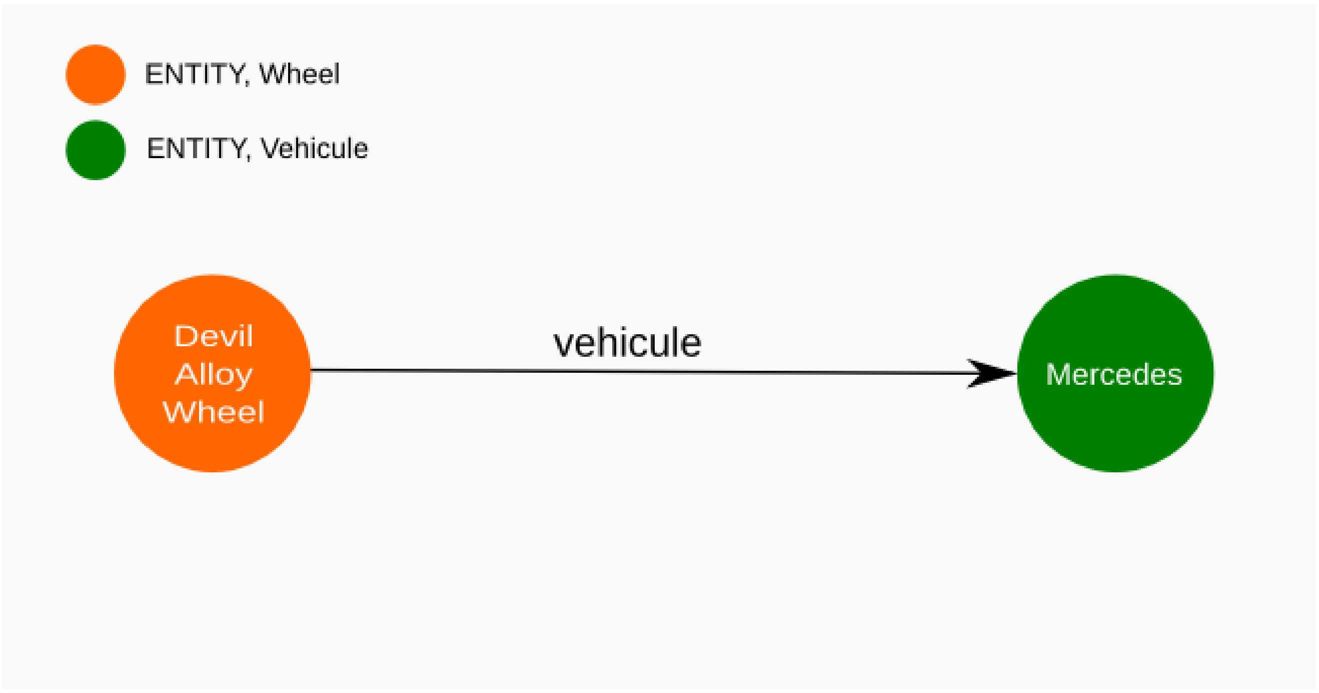


Figure 12.7.

Example 12.8. Bidirectional one-to-one

```
@Entity
public class Husband {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Wife wife;
    // getters, setters ...
}

@Entity
public class Wife {

    @Id
    private String id;
    private String name;

    @OneToOne(mappedBy = "wife")
    private Husband husband;

    // getters, setters ...
}
```

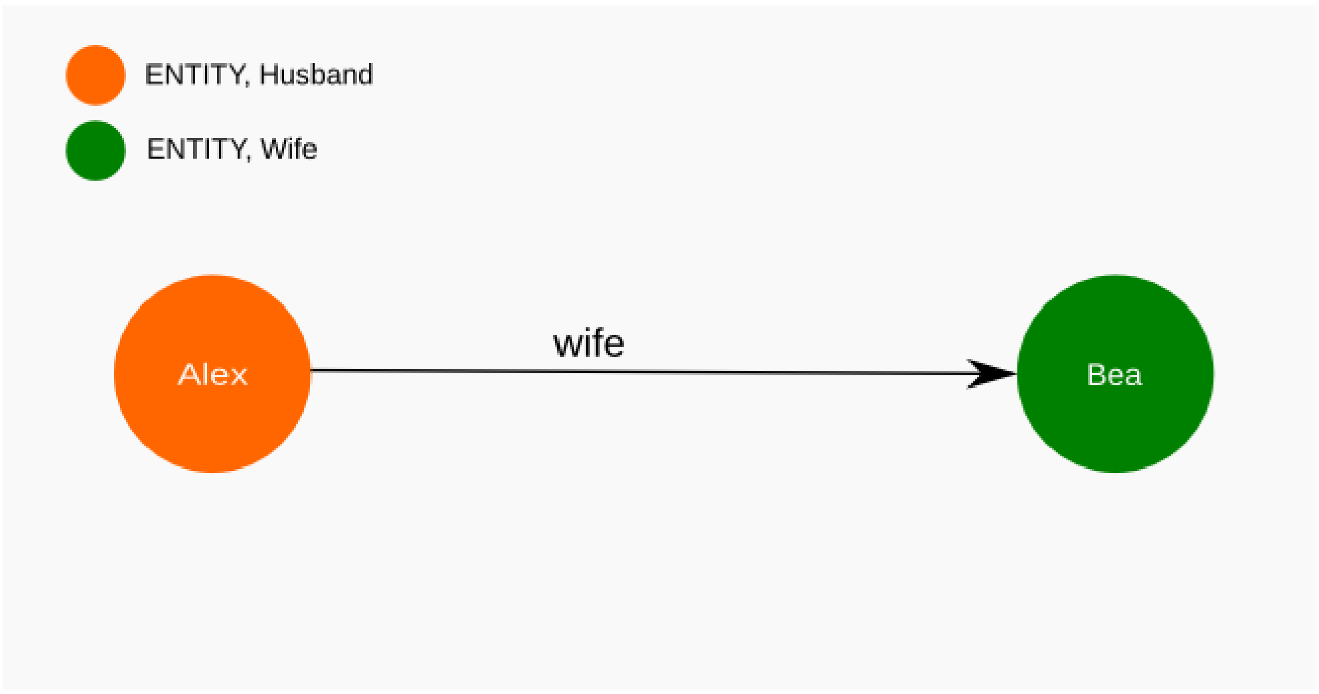


Figure 12.8.

Example 12.9. Unidirectional one-to-many

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();
    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

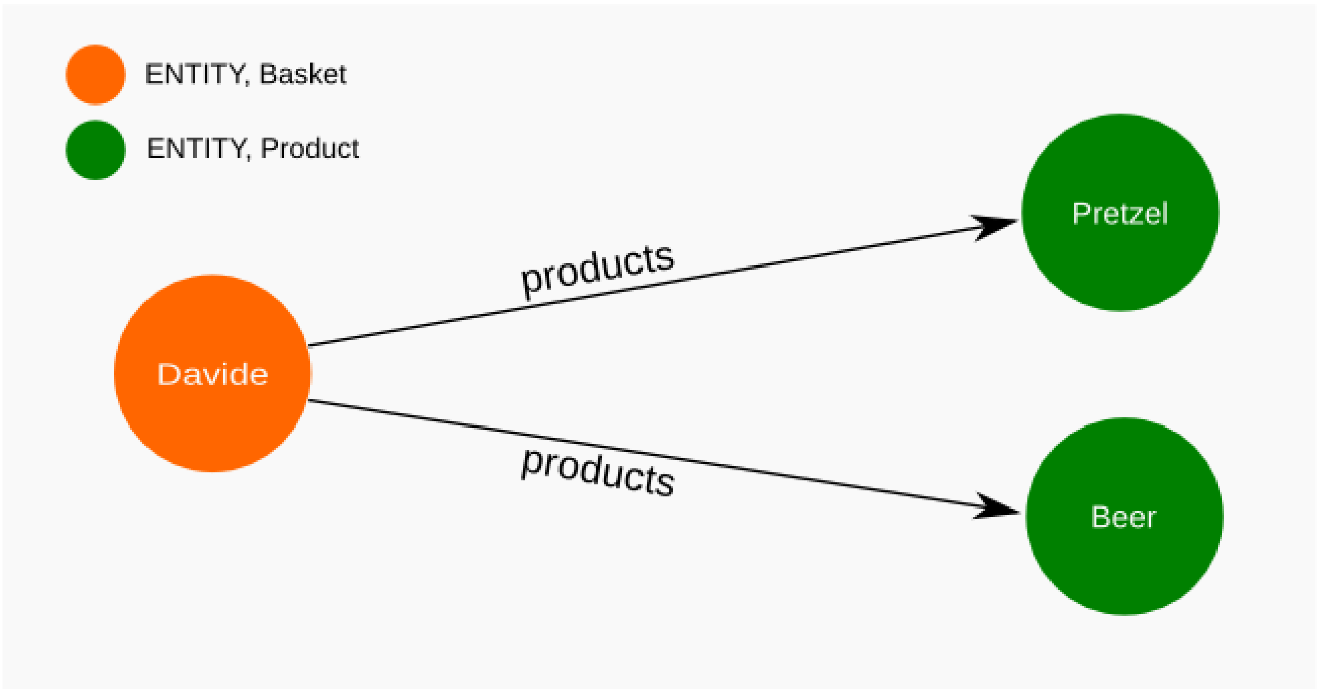


Figure 12.9.

Example 12.10. Unidirectional one-to-many using maps with defaults

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}
```

```
@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

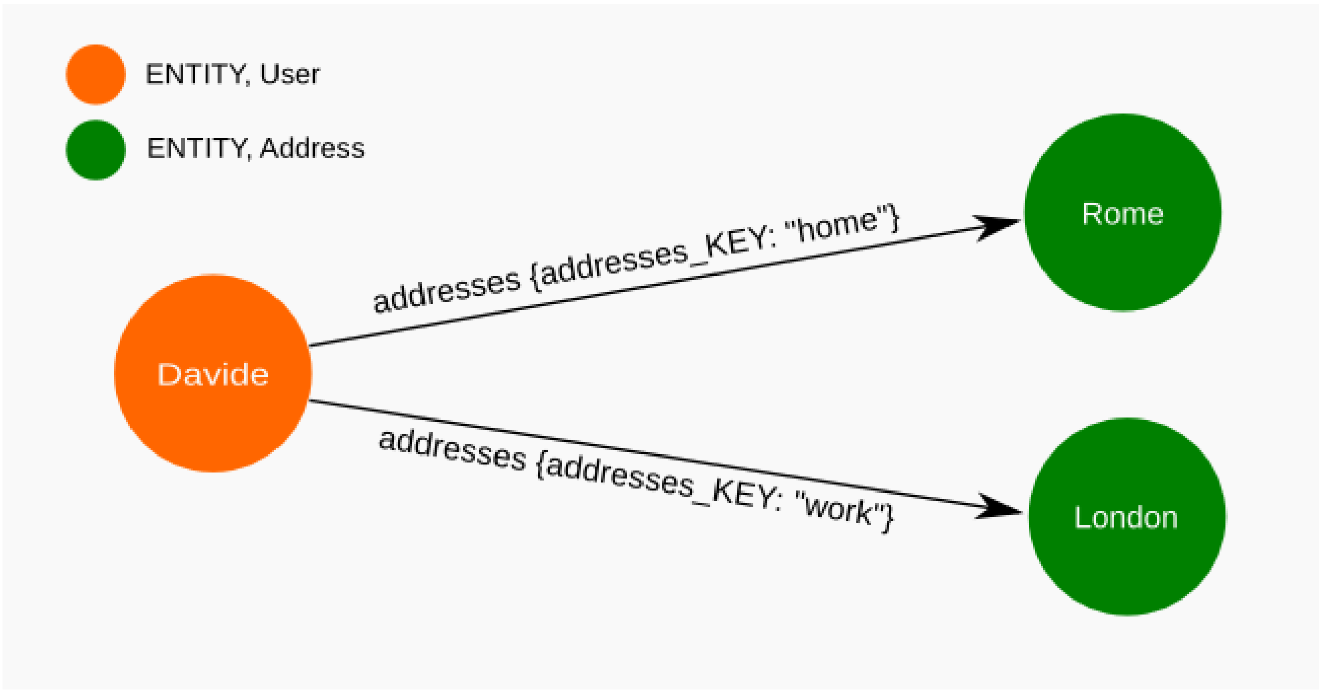

ENTITY, User
ENTITY, Address
Davide
Rome
London
addresses {addresses_KEY: "home"}
addresses {addresses_KEY: "work"}

Figure 12.10.

Example 12.11. Unidirectional one-to-many using maps with @MapKeyColumn

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    @MapKeyColumn(name = "addressType")
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {
    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

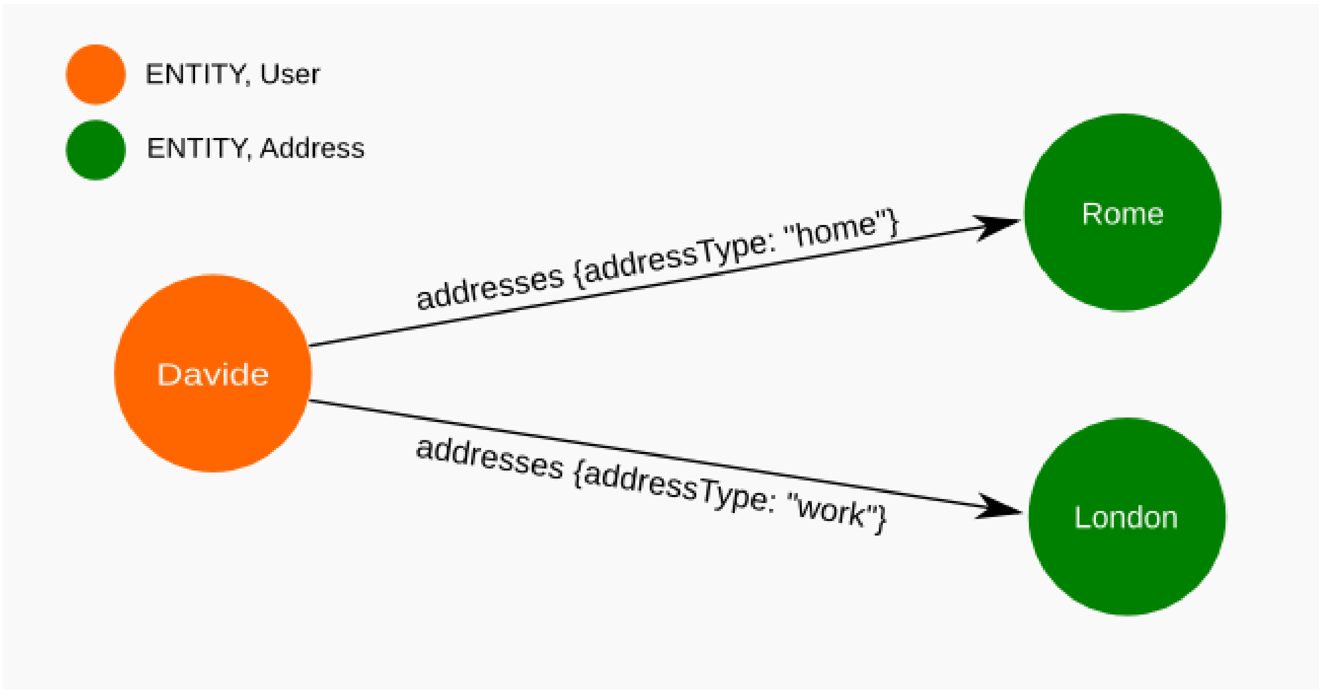

ENTITY, User
ENTITY, Address
Rome
Davide
London
addresses {addressType: "home"}
addresses {addressType: "work"}

Figure 12.11.

Example 12.12. Unidirectional many-to-one

```
@Entity
public class JavaUserGroup {

    @Id
    private String jug_id;
    private String name;

    // getters, setters ...
}

@Entity
public class Member {

    @Id
    private String id;
    private String name;

    @ManyToOne
    private JavaUserGroup memberOf;
    // getters, setters ...
}
```

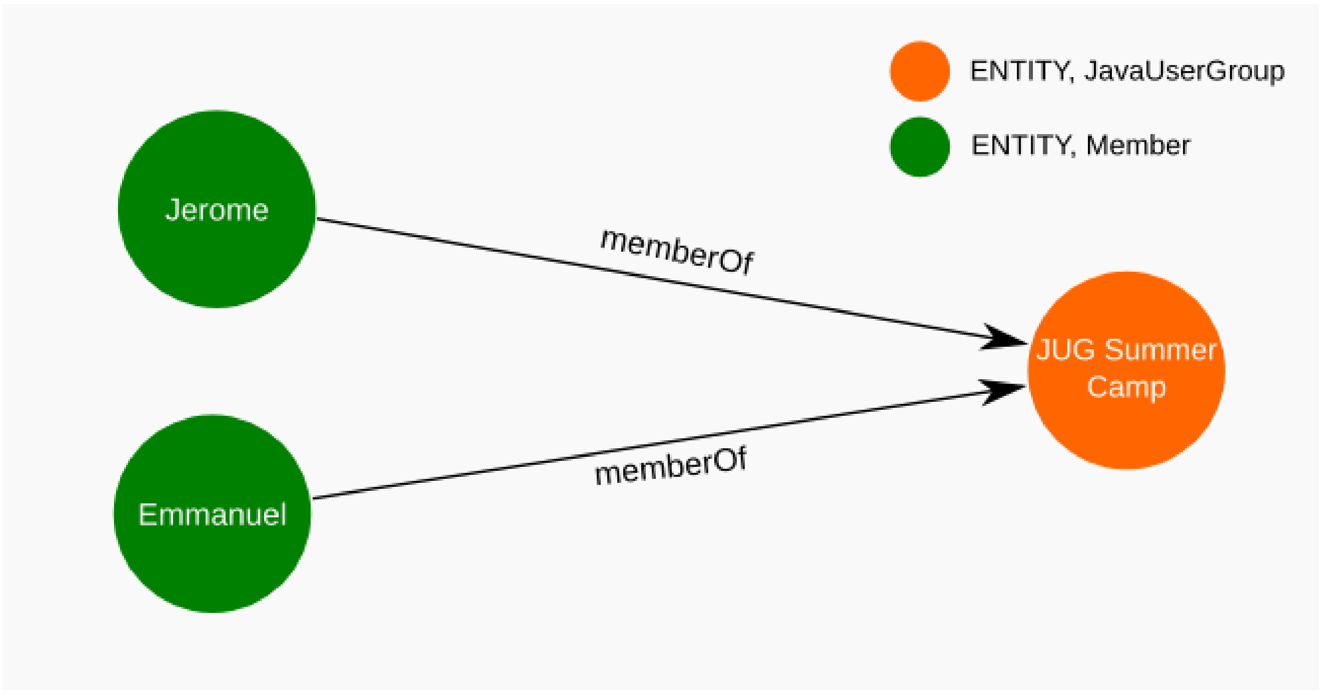


Figure 12.12.

Example 12.13. Bidirectional many-to-one

```
@Entity
public class SalesForce {

    @Id
    private String id;
    private String corporation;

    @OneToMany(mappedBy = "salesForce")
    private Set<SalesGuy> salesGuys = new HashSet<SalesGuy>();

    // getters, setters ...
}

@Entity
public class SalesGuy {
    private String id;
    private String name;

    @ManyToOne
    private SalesForce salesForce;

    // getters, setters ...
}
```

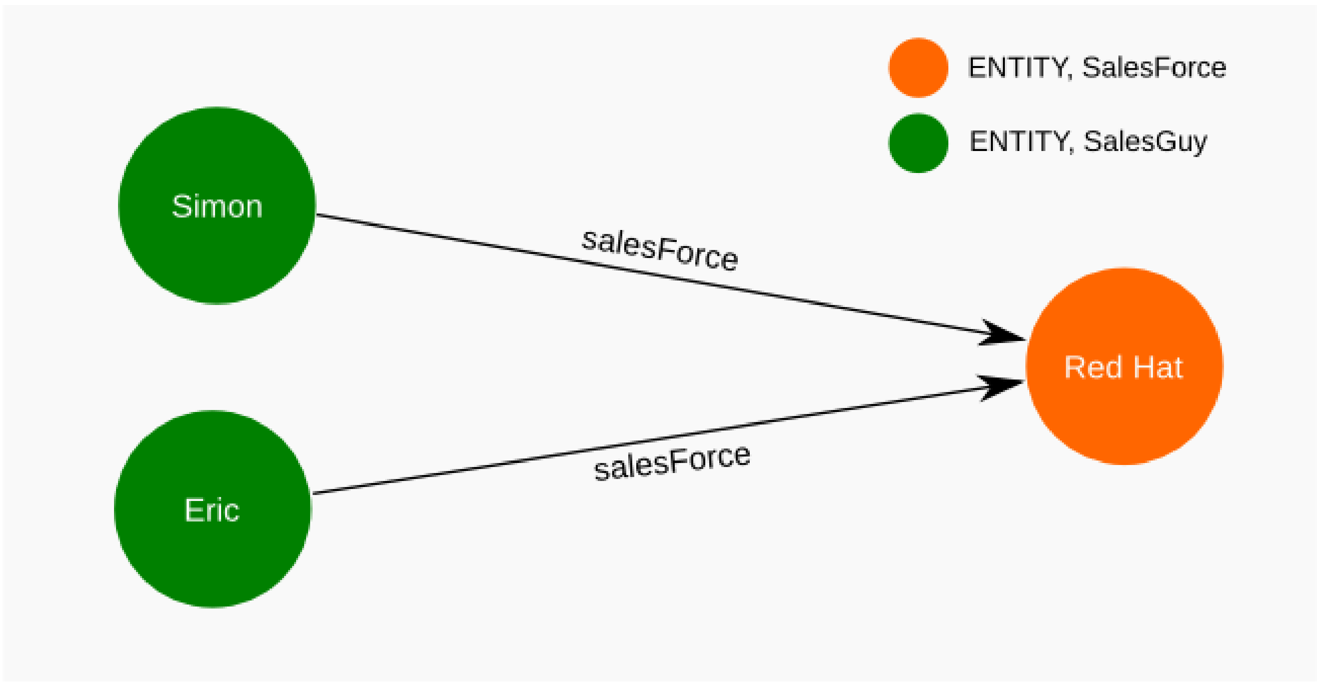


Figure 12.13.

Example 12.14. Unidirectional many-to-many

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

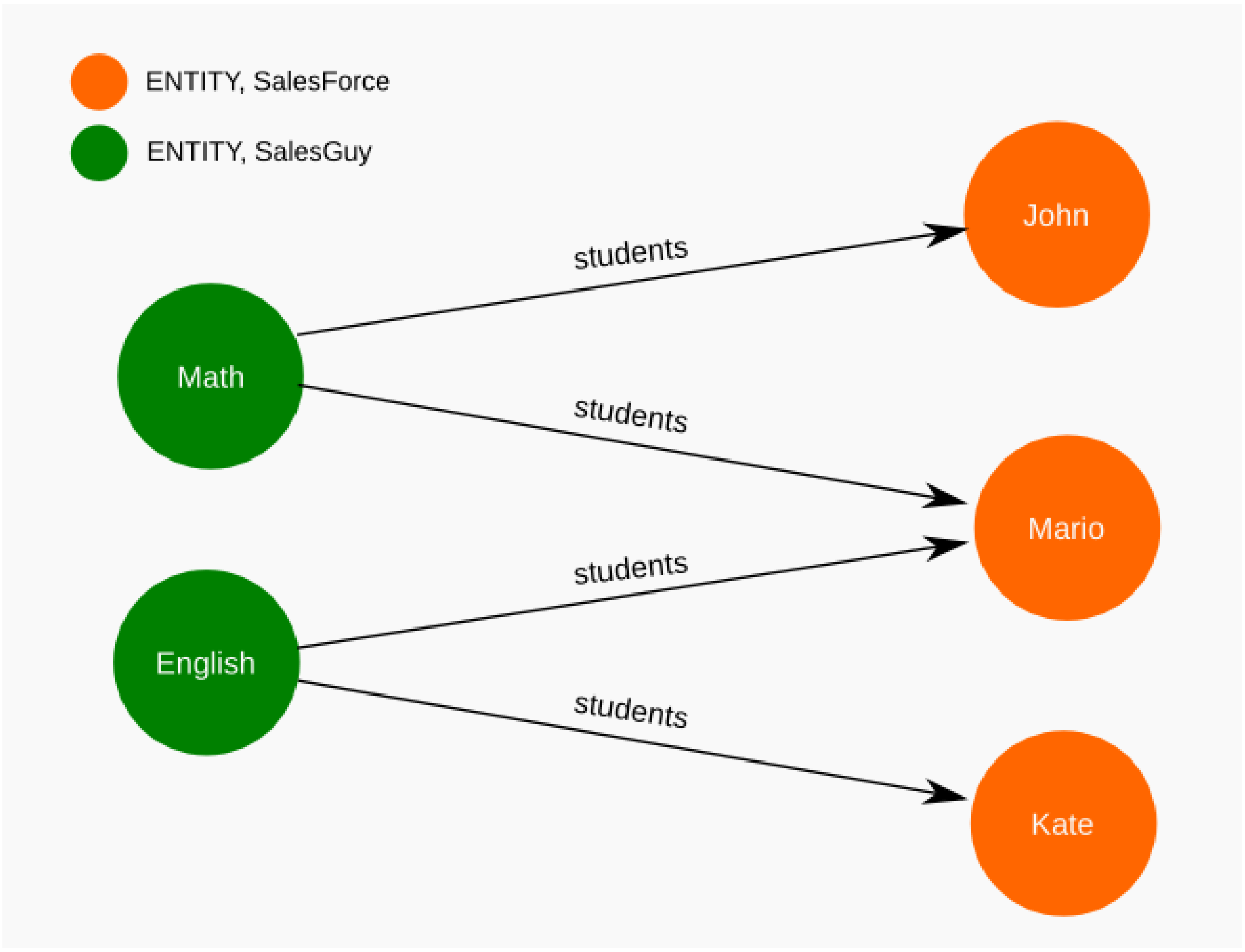


Figure 12.14.

Example 12.15. Bidirectional many-to-many

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany( mappedBy = "bankAccounts" )
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

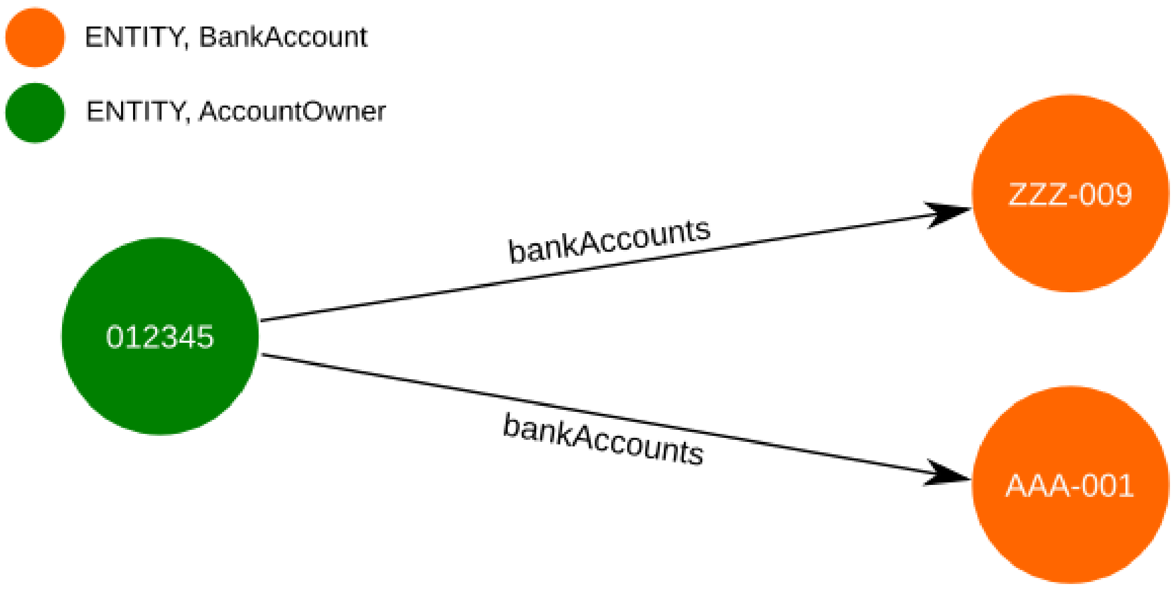


Figure 12.15.

12.3.4. Auto-generated Values

Hibernate OGM supports the table generation strategy as well as the sequence generation strategy with Neo4j. It is generally recommended to work with the latter, as it allows a slightly more efficient querying for the next sequence value.

Sequence-based generators are represented by nodes in the following form:

Example 12.16. GenerationType.SEQUENCE

```
@Entity
public class Song {

    ...

    @Id
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "songSequenceGenerator" )
    @SequenceGenerator(
            name = "songSequenceGenerator",
            sequenceName = "song_sequence",
            initialValue = INITIAL_VALUE,
            allocationSize = 10)
    public Long getId() {
        return id;
    }

    ...
```

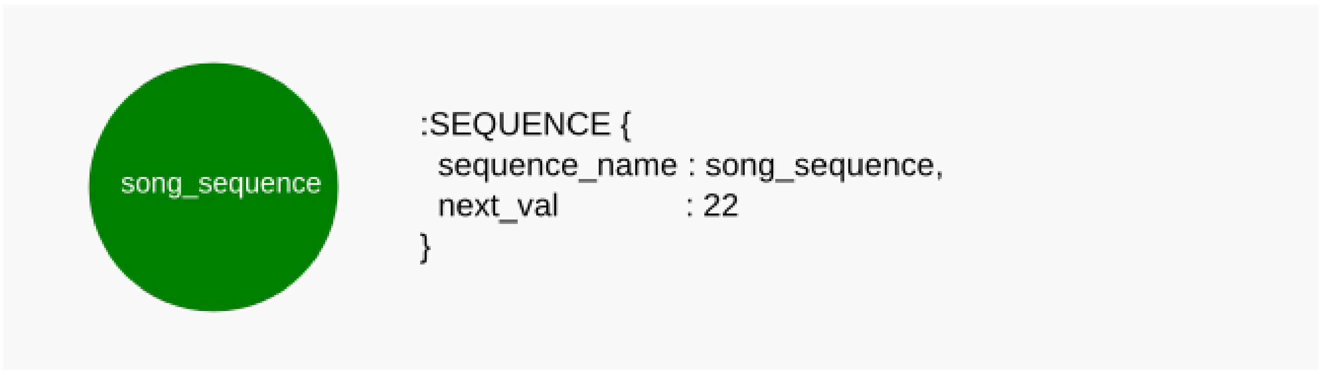


Figure 12.16.

Each sequence generator node is labelled with `SEQUENCE`. The sequence name can be specified via `@SequenceGenerator#sequenceName()`. A unique constraint is applied to the property `sequence_name` in order to ensure uniqueness of sequences.

If required, you can set the initial value of a sequence and the increment size via `@SequenceGenerator#initialValue()` and `@SequenceGenerator#allocationSize()`, respectively. The options `@SequenceGenerator#catalog()` and `@SequenceGenerator#schema()` are not supported.

Table-based generators are represented by nodes in the following form:

Example 12.17. GenerationType.TABLE

```
@Entity
public class Video {

    ...

    @Id
    @GeneratedValue( strategy = GenerationType.TABLE, generator = "video" )
    @TableGenerator(
         name = "video",
         table = "Sequences",
         pkColumnName = "key",
         pkColumnValue = "video",
         valueColumnName = "seed"
    )
    public Integer getId() {
        return id;
    }

    ...
```

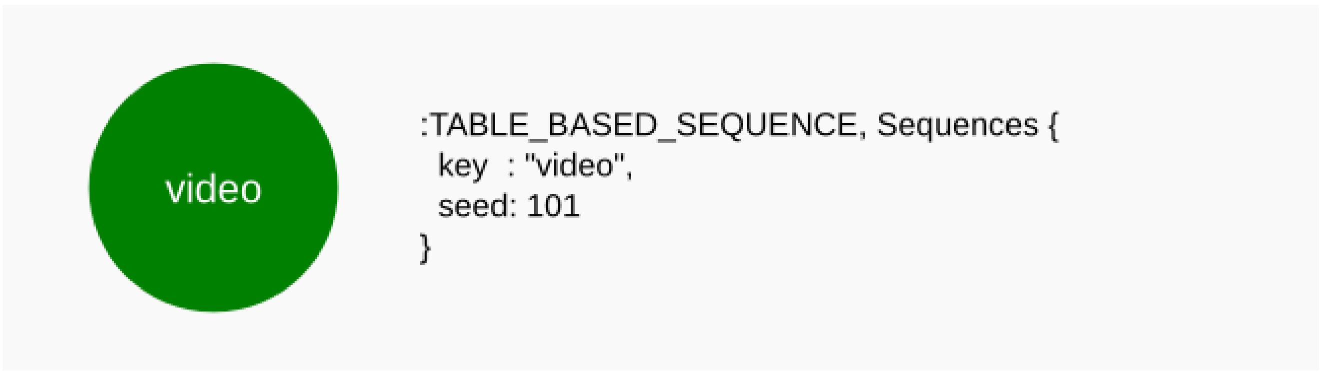


Figure 12.17.

Each table generator node is labelled with `TABLE_BASED_SEQUENCE` and the table name as specified via `@TableGenerator#table()`. The sequence name is to be given via `@TableGenerator#pkColumnValue()`. The node properties holding the sequence name and value can be configured via `@TableGenerator#pkColumnName()` and `@TableGenerator#valueColumnName()`, respectively. A unique constraint is applied to the property `sequence_name` to avoid the same sequence name is used twice within the same "table".

If required, you can set the initial value of a sequence and the increment size via `@TableGenerator#initialValue()` and `@TableGenerator#allocationSize()`, respectively. The options `@TableGenerator#catalog()`, `@TableGenerator#schema()`, `@TableGenerator#uniqueConstraints()` and `@TableGenerator#indexes()` are not supported.

12.3.5. Labels summary

The maximum number of labels the database can contain is roughly 2 billion.

The following summary will help you to keep track of the labels assigned to a new node:

Table 12.1. Summary of the labels assigned to a new node

NODE TYPE	LABELS
Entity	ENTITY, <Entity class name>
Embeddable	EMBEDDED, <Embeddable class name>
GenerationType.SEQUENCE	SEQUENCE
GenerationType.TABLE	TABLE_BASED_SEQUENCE, <Table name>

12.4. Transactions

Caution

Neo4J operations must be executed inside a transaction. Make sure your interactions with Hibernate OGM are within a transaction when you target Neo4J.

Unless a different `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform` is specified, Hibernate OGM will use a specific implementation to integrate with the Neo4j transaction mechanism. This means that you can start and commit transaction using the Hibernate session.

The drawback is that it is not possible at the moment to let Neo4j participate in managed JTA transactions spanning several resources (see issue *OGM-370* [https://hibernate.atlassian.net/browse/OGM-370]).

Example 12.18. Example of starting and committing transactions

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

Account account = new Account();
account.setLogin( "myAccount" );
session.persist( account );

tx.commit();

...

tx = session.beginTransaction();
Account savedAccount =  (Account) session.get( Account.class, account.getId() );
tx.commit();
```

12.5. Queries

You can express queries in a few different ways:

- using JP-QL
- using the Cypher query language

Note

Neo4J makes use of a Lucene version which is not compatible with the most recent Hibernate Search version. This unfortunately makes it impossible to use the latest Hibernate Search version and Neo4J embedded in the same application.

While you can use JP-QL for simple queries, you might hit limitations. The current recommended approach is to use native Cypher queries if your query involves nested (list of) elements.

12.5.1. JP-QL queries

Hibernate OGM is a work in progress, so only a sub-set of JP-QL constructs is available when using the JP-QL query support. This includes:

- simple comparisons using "<", "#", "=", ">=" and ">"
- `IS NULL` and `IS NOT NULL`
- the boolean operators `AND`, `OR`, `NOT`
- `LIKE`, `IN` and `BETWEEN`
- `ORDER BY`

Queries using these constructs will be transformed into equivalent *Cypher queries* [http://docs.neo4j.org/chunked/stable/cypher-query-lang.html].

Note

Let us know *by opening an issue or sending an email* what query you wish to execute. Expanding our support in this area is high on our priority list.

12.5.2. Cypher queries

Hibernate OGM also supports *Cypher queries* [http://docs.neo4j.org/chunked/stable/cypher-query-lang.html] for Neo4j. You can execute Cypher queries as shown in the following example:

Example 12.19. Using the JPA API

```
@Entity
public class Poem {

    @Id
    private Long id;

    private String name;

    private String author;

   // getters, setters ...

}

...
```

```
javax.persistence.EntityManager em = ...

// a single result query
String query1 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) RETURN n";
Poem poem = (Poem) em.createNativeQuery( query1, Poem.class ).getSingleResult();

// query with order by
String query2 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) " +
                "RETURN n ORDER BY n.name";
List<Poem> poems = em.createNativeQuery( query2, Poem.class ).getResultList();

// query with projections
String query3 = MATCH ( n:Poem ) RETURN n.name, n.author ORDER BY n.name";
List<Object[]> poemNames = (List<Object[]>)em.createNativeQuery( query3 )
                               .getResultList();
```

The result of a query is a managed entity (or a list thereof) or a projection of attributes in form of an object array, just like you would get from a JP-QL query.

Example 12.20. Using the Hibernate native API

```
OgmSession session = ...

String query1 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) " +
                "RETURN n";
Poem poem = session.createNativeQuery( query1 )
                      .addEntity( "Poem", Poem.class )
                      .uniqueResult();

String query2 = "MATCH ( n:Poem { name:'Portia', author:'Oscar Wilde' } ) " +
                "RETURN n ORDER BY n.name";
List<Poem> poems = session.createNativeQuery( query2 )
                      .addEntity( "Poem", Poem.class )
                      .list();
```

Native queries can also be created using the `@NamedNativeQuery` annotation:

Example 12.21. Using @NamedNativeQuery

```
@Entity
@NamedNativeQuery(
   name = "AthanasiaPoem",
   query = "MATCH ( n:Poem { name:'Athanasia', author:'Oscar Wilde' } ) RETURN n",
   resultClass = Poem.class )
public class Poem { ... }

...

// Using the EntityManager
Poem poem1 = (Poem) em.createNamedQuery( "AthanasiaPoem" )
                     .getSingleResult();
```

```
// Using the Session
Poem poem2 = (Poem) session.getNamedQuery( "AthanasiaPoem" )
                     .uniqueResult();
```

Hibernate OGM stores data in a natural way so you can still execute queries using your favorite tool, the main drawback is that the results are going to be raw Neo4j elements and not managed entities.

CouchDB (Experimental)

CouchDB [https://couchdb.apache.org/] is a document-oriented datastore which stores your data in form of JSON documents and exposes its API via HTTP based on REST principles. It is thus very easy to access from a wide range of languages and applications.

Note

Support for CouchDB is considered an EXPERIMENTAL feature as of this release. In particular you should be prepared for possible changes to the persistent representation of mapped objects in future releases.

Also be aware of the fact that partial updates are unsupported at the moment (*OGM-388* [https://hibernate.atlassian.net/browse/OGM-388]). Instead always the entire document will be replaced during updates. This means that fields possibly written by other applications but not mapped to properties in your domain model will get lost.

The `ASSOCIATION_DOCUMENT` mode for storing associations should be used with care as there is potential for lost updates (*OGM-461* [https://hibernate.atlassian.net/browse/OGM-461]). It is recommended to use the `IN_ENTITY` mode (which is the default).

Should you find any bugs or have feature requests for this dialect, then please open a ticket in the *OGM issue tracker* [https://hibernate.atlassian.net/browse/OGM].

13.1. Configuring CouchDB

Hibernate OGM uses the excellent *RESTEasy* [https://www.jboss.org/resteasy] library to talk to CouchDB stores, so there is no need to include any of the Java client libraries for CouchDB in your classpath.

The following properties are available to configure CouchDB support in Hibernate OGM:

CouchDB datastore configuration properties

hibernate.ogm.datastore.provider
: To use CouchDB as a datastore provider, this property must be set to `couchdb_experimental`

hibernate.ogm.option.configurator
: The fully-qualified class name or an instance of a programmatic option configurator (see *Section 13.1.2, “Programmatic configuration”*)

hibernate.ogm.datastore.host
: The hostname of the CouchDB instance. The default value is `127.0.0.1`.

hibernate.ogm.datastore.port
: The port used by the CouchDB instance. The default value is `5984`.

hibernate.ogm.datastore.database
: The database to connect to. This property has no default value.

hibernate.ogm.datastore.create_database
: Whether to create the specified database in case it does not exist or not. Can be `true` or `false` (default). Note that the specified user must have the right to create databases if set to `true`.

hibernate.ogm.datastore.username
: The username used when connecting to the CouchDB server. Note that this user must have the right to create design documents in the chosen database. This property has no default value. Hibernate OGM currently does not support accessing CouchDB via HTTPS; if you're interested in such functionality, let us know.

hibernate.ogm.datastore.password
: The password used to connect to the CouchDB server. This property has no default value. This property is ignored if the username isn't specified.

hibernate.ogm.datastore.document.association_storage
: Defines the way OGM stores association information in CouchDB. The following two strategies exist (values of the `org.hibernate.ogm.datastore.document.options.AssociationStorageType` enum): `IN_ENTITY` (store association information within the entity) and `ASSOCIATION_DOCUMENT` (store association information in a dedicated document per association). `IN_ENTITY` is the default and recommended option unless the association navigation data is much bigger than the core of the document and leads to performance degradation.

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `CouchDBProperties` when specifying the configuration properties listed above. Common properties shared between (document) stores are declared on `OgmProperties` and `DocumentStoreProperties`, respectively. To ease migration between stores, it is recommended to reference these constants directly from there.

13.1.1. Annotation based configuration

Hibernate OGM allows to configure store-specific options via Java annotations. When working with the CouchDB backend, you can specify how associations should be stored using the `AssociationStorage` annotation (refer to *Section 13.2, "Storage principles"* to learn more about association storage strategies in general).

The following shows an example:

Example 13.1. Configuring the association storage strategy using annotations

```
@Entity
@AssociationStorage(AssociationStorageType.ASSOCIATION_DOCUMENT)
public class Zoo {

    @OneToMany
    private Set<Animal> animals;

    @OneToMany
    private Set<Person> employees;

    @OneToMany
    @AssociationStorage(AssociationStorageType.IN_ENTITY)
    private Set<Person> visitors;

    //...
}
```

The annotation on the entity level expresses that all associations of the `Zoo` class should be stored in separate assocation documents. This setting applies to the `animals` and `employees` associations. Only the elements of the `visitors` association will be stored in the document of the corresponding `Zoo` entity as per the configuration of that specific property which takes precedence over the entity-level configuration.

13.1.2. Programmatic configuration

In addition to the annotation mechanism, Hibernate OGM also provides a programmatic API for applying store-specific configuration options. This can be useful if you can’t modify certain entity types or don’t want to add store-specific configuration annotations to them. The API allows set options in a type-safe fashion on the global, entity and property levels.

When working with CouchDB, you can currently configure the following options using the API:

- association storage strategy (on the global, entity and property level)

To set this option via the API, you need to create an `OptionConfigurator` implementation as shown in the following example:

Example 13.2. Example of an option configurator

```
public class MyOptionConfigurator extends OptionConfigurator {

    @Override
    public void configure(Configurable configurable) {
        configurable.configureOptionsFor( CouchDB.class )
            .associationStorage( AssociationStorageType.ASSOCIATION_DOCUMENT )
            .entity( Zoo.class )
```

```
                .property( "visitors", ElementType.FIELD )
                    .associationStorage( AssociationStorageType.IN_ENTITY )
            .entity( Animal.class )
                .associationStorage( AssociationStorageType.ASSOCIATION_DOCUMENT );
    }
}
```

The call to `configureOptionsFor()`, passing the store-specific identifier type `CouchDB`, provides the entry point into the API. Following the fluent API pattern, you then can configure global options and navigate to single entities or properties to apply options specific to these.

Options given on the property level precede entity-level options. So e.g. the `visitors` association of the `Zoo` class would be stored using the in entity strategy, while all other associations of the `Zoo` entity would be stored using separate association documents.

Similarly, entity-level options take precedence over options given on the global level. Global-level options specified via the API complement the settings given via configuration properties. In case a setting is given via a configuration property and the API at the same time, the latter takes precedence.

Note that for a given level (property, entity, global), an option set via annotations is overridden by the same option set programmatically. This allows you to change settings in a more flexible way if required.

To register an option configurator, specify its class name using the `hibernate.ogm.option.configurator` property. When bootstrapping a session factory or entity manager factory programmatically, you also can pass in an `OptionConfigurator` instance or the class object representing the configurator type.

13.2. Storage principles

Hibernate OGM tries to make the mapping to the underlying datastore as natural as possible so that third party applications not using Hibernate OGM can still read and update the same datastore. The following describe how entities and associations are mapped to CouchDB documents by Hibernate OGM.

13.2.1. Properties and built-in types

Note

Hibernate OGM doesn't store null values in CouchDB, setting a value to null will be the same as removing the field in the corresponding object in the db.

Hibernate OGM support by default the following types:

- `java.lang.String`

```
{ "text" : "Hello world!" }
```

- `java.lang.Character` (or char primitive)

```
{ "delimiter" : "/" }
```

- `java.lang.Boolean` (or boolean primitive)

```
{ "favorite" : true }
```

- `java.lang.Byte` (or byte primitive)

```
{ "display_mask" : "70" }
```

- `java.lang.Short` (or short primitive)

```
{ "urlPort" : 80 }
```

- `java.lang.Integer` (or int primitive)

```
{ "stockCount" : 12309 }
```

- `java.lang.Long` (or long primitive)

```
{ "userId" : "-6718902786625749549" }
```

- `java.lang.Float` (or float primitive)

```
{ "visitRatio" : 10.4 }
```

- `java.lang.Double` (or double primitive)

```
{ "tax_percentage" : 12.34 }
```

- `java.math.BigDecimal`

```
{ "site_weight" : "21.77" }
```

- `java.math.BigInteger`

```
{ "site_weight" : "444" }
```

- `java.util.Calendar`

```
{ "creation" : "2014-11-18T15:51:26.252Z" }
```

- `java.util.Date`

```
{ "last_update" : "2014-11-18T15:51:26.252Z" }
```

- `java.util.UUID`

```
{ "serialNumber" : "71f5713d-69c4-4b62-ad15-aed8ce8d10e0" }
```

- `java.util.URL`

```
{ "url" : "http://www.hibernate.org/" }
```

13.2.2. Entities

Entities are stored as CouchDB documents and not as BLOBs which means each entity property will be translated into a document field. You can use the name property of the `@Table` and `@Column` annotations to rename the collections and the document's fields if you need to.

CouchDB provides a built-in mechanism for detecting concurrent updates to one and the same document. For that purpose each document has an attribute named `_rev` (for "revision") which is to be passed back to the store when doing an update. So when writing back a document and the document's revision has been altered by another writer in parallel, CouchDB will raise an optimistic locking error (you could then e.g. re-read the current document version and try another update).

For this mechanism to work, you need to declare a property for the `_rev` attribute in all your entity types and mark it with the `@Version` and `@Generated` annotations. The first marks it as a property used for optimistic locking, while the latter advices Hibernate OGM to refresh that property after writes since its value is managed by the datastore.

Warning

Not mapping the `_rev` attribute may cause lost updates, as Hibernate OGM needs to re-read the current revision before doing an update in this case. Thus a warning will be issued during initialization for each entity type which fails to map that property.

The following shows an example of an entity and its persistent representation in CouchDB.

Example 13.3. Example of an entity and its representation in CouchDB

```
@Entity
public class News {

    @Id
    private String id;

    @Version
    @Generated
    @Column(name="_rev")
    private String revision;

    private String title;

    private String description;

    //getters, setters ...
}
```

```
{
    "_id": "News:id_:news-1_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "News",
    "title": "On the merits of NoSQL",
    "description": "This paper discuss why NoSQL will save the world for good"
}
```

Note that CouchDB doesn't have a concept of "tables" or "collections" as e.g. MongoDB does; Instead all documents are stored in one large bucket. Thus Hibernate OGM needs to add two additional attributes: `$type` which contains the type of a document (entity vs. association documents) and `$table` which specifies the entity name as derived from the type or given via the `@Table` annotation.

> **Note**
>
> Attributes whose name starts with the "$" character are managed by Hibernate OGM and thus should not be modified manually. Also it is not recommended to start the names of your attributes with the "$" character to avoid collisions with attributes possibly introduced by Hibernate OGM in future releases.

Example 13.4. Rename field and collection using @Table and @Column

```
@Entity
@Table(name="Article")
public class News {

    @Id
    @Column(name="code")
    private String id;

    @Version
    @Generated
    @Column(name="_rev")
    private String revision;

    private String title;

    @Column(name="desc")
    private String description;

    //getters, setters ...
}
```

```
{
    "_id": "Article:code_:news-1_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "Article",
    "title": "On the merits of NoSQL",
    "desc": "This paper discuss why NoSQL will save the world for good"
}
```

13.2.2.1. Identifiers

The `_id` field of a CouchDB document is directly used to store the identifier columns mapped in the entities. You can use any persistable Java type as identifier type, e.g. `String` or `long`.

Hibernate OGM will convert the `@Id` property into a `_id` document field so you can name the entity id like you want, it will always be stored into `_id`.

Note that you also can work with embedded ids (via `@EmbeddedId`), but be aware of the fact that CouchDB doesn't support storing embedded structures in the `_id` attribute. Hibernate OGM thus will create a concatenated representation of the embedded id's properties in this case.

Example 13.5. Entity with @EmbeddedId

```
@Entity
public class News {

    @EmbeddedId
    private NewsID newsId;

    // getters, setters ...
}

@Embeddable
public class NewsID implements Serializable {

    private String title;
    private String author;

    // getters, setters ...
}
```

```
{
    "_id": "News:newsId.author_newsId.title_:Guillaume_How to use Hibernate OGM ?_",
    "_rev": "2-1f02af4fabba7b4fa7394f1167244226",
    "$type": "entity",
    "$table": "News",
    "newsId": {
        "author": "Guillaume",
        "title": "How to use Hibernate OGM ?"
    }
}
```

13.2.2.2. Identifier generation strategies

You can assign id values yourself or let Hibernate OGM generate the value using the `@GeneratedValue` annotation.

Two main strategies are supported:

1. *TABLE*

2. *SEQUENCE*

Both strategy will create a new document containg the next value to use for the id, the difference between the two strategies is the name of the field containing the values.

Hibernate OGM goes not support the `IDENTITY` strategy and an exception is thrown at startup when it is used. The `AUTO` strategy is the same as the *SEQUENCE* one.

1) TABLE generation strategy

Example 13.6. Id generation strategy TABLE using default values

```
@Entity
public class Video {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private Integer id;
    private String name

    // getters, setters, ...
}
```

```
{
   "_id": "Video:id_:1_",
   "_rev": "1-b4c16b6cd8a083f2173f8df19bd24750",
   "$type": "entity",
   "$table": "Video",
   "id": 1,
   "name": "Scream",
   "director": "Wes Craven"
}
```

```
{
   "_id": "hibernate_sequences:sequence_name:default",
   "_rev": "1-ebb82f1cea26d57f47a290fb0c1cc58f",
   "$type": "sequence",
   "next_val": "2"
}
```

Example 13.7. Id generation strategy TABLE using a custom table

```
@Entity
public class Video {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "video")
    @TableGenerator(
            name = "video",
            table = "sequences",
            pkColumnName = "key",
            pkColumnValue = "video",
            valueColumnName = "seed"
    )
    private Integer id;

    private String name;

    // getter, setters, ...
}
```

```
@Entity
public class Video {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "video")
    @TableGenerator(
            name = "video",
            table = "sequences",
            pkColumnName = "key",
            pkColumnValue = "video",
            valueColumnName = "seed"
    )
    private Integer id;
    private String name

    // getters, setters, ...
}
```

```
{
   "_id": "sequences:key:video",
   "_rev": "2-78b3450e0658743164828c4076e06a49",
   "$type": "sequence",
   "seed": "101"
}
```

2) SEQUENCE generation strategy

Example 13.8. SEQUENCE id generation strategy using default values

```
@Entity
public class Song {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Long id;

  private String title;

  // getters, setters ...
}
```

```
{
   "_id": "Song:id_:2_",
   "_rev": "1-63bc100449fb2840067028c3825ed784",
   "$type": "entity",
   "$table": "Song",
   "id": "2",
   "title": "Ave Maria",
   "singer": "Charlotte Church"
}
```

```
{
   "_id": "hibernate_sequences:sequence_name:hibernate_sequence",
   "_rev": "2-dcc622bcb1389ad18829dcfc8b812c87",
   "$type": "sequence",
   "next_val": "3"
}
```

Example 13.9. SEQUENCE id generation strategy using custom values

```
@Entity
public class Song {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "songSequenceGenerator")
  @SequenceGenerator(
      name = "songSequenceGenerator",
      sequenceName = "song_sequence",
      initialValue = 2,
      allocationSize = 20
  )
  private Long id;

  private String title;

  // getters, setters ...
}
```

```
{
   "_id": "Song:id_:2_",
   "_rev": "1-63bc100449fb2840067028c3825ed784",
   "$type": "entity",
   "$table": "Song",
   "id": "2",
   "title": "Ave Maria",
   "singer": "Charlotte Church"
}
```

```
{
   "_id": "hibernate_sequences:sequence_name:song_sequence",
   "_rev": "2-df47883f076c84cb953f9184de7aa82a",
   "$type": "sequence",
   "next_val": "21"
}
```

13.2.2.3. Embedded objects and collections

Hibernate OGM stores elements annotated with `@Embedded` or `@ElementCollection` as nested documents of the owning entity.

Example 13.10. Embedded object

```
@Entity
public class News {

    @Id
    private String id;
    private String title;

    @Embedded
    private NewsPaper paper;

    // getters, setters ...
}

@Embeddable
public class NewsPaper {

    private String name;
    private String owner;

    // getters, setters ...
}
```

```
{
    "_id": "News:id_:939c892d-1129-4aff-abf8-e6c26e59dcb_",
    "_rev": "2-1f02af4fabba7b4fa7394f1167244226",
    "$type": "entity",
    "$table": "News",
    "id": "939c892d-1129-4aff-abf8-e6c26e59dcb",
    "paper": {
        "name": "NoSQL journal of prophecies",
        "owner": "Delphy"
    }
}
```

Example 13.11. @ElementCollection with primitive types

```
@Entity
public class AccountWithPhone {
    @Id
    private String id;

    @ElementCollection
    private List<String> mobileNumbers;

    // getters, setters ...
}
```

AccountWithPhone collection

```
{
   "_id": "AccountWithPhone:id_:2_",
   "_rev": "2-a71f7c0d621a08232568f9840bff05ce",
   "$type": "entity",
   "$table": "AccountWithPhone",
   "id": "2",
   "mobileNumbers": [
       "+1-222-555-0222",
       "+1-202-555-0333"
   ]
}
```

Example 13.12. @ElementCollection with one attribute

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

```
{
    "_id": "grandmother:id_:86ada718-f2a2-4299-b6ac-3d90b1ef2331_",
    "_rev": "2-1f02af4fabba7b4fa7394f1167244226",
    "$type": "entity",
    "$table": "grandmother",
    "id": "86ada718-f2a2-4299-b6ac-3d90b1ef2331",
    "grandChildren" : [ "Luke", "Leia" ]
}
```

The class `GrandChild` has only one attribute `name`, this means that Hibernate OGM doesn't need to store the name of the attribute.

If the nested document has two or more fields, like in the following example, Hibernate OGM will store the name of the fields as well.

Example 13.13. @ElementCollection with @OrderColumn

```
@Entity
public class GrandMother {

    @Id
    private String id;

    @ElementCollection
    @OrderColumn( name = "birth_order" )
    private List<GrandChild> grandChildren = new ArrayList<GrandChild>();

    // getters, setters ...
}

@Embeddable
public class GrandChild {

    private String name;

    // getters, setters ...
}
```

```
{
    "_id": "GrandMother:id_:86ada718-f2a2-4299-b6ac-3d90b1ef2331_",
    "_rev": "2-1f02af4fabba7b4fa7394f1167244226",
    "$type": "entity",
    "$table": "GrandMother",
    "grandChildren" : [
            {
                "name" : "luke",
                "birth_order" : 0
            },
            {
                "name" : "leia",
                "birthorder" : 1
            }
    ]
}
```

13.2.3. Associations

Hibernate OGM CouchDB provides two strategies to store navigation information for associations:

- `IN_ENTITY` (default)
- `ASSOCIATION_DOCUMENT`

You can switch between the two strategies using:

- the `@AssociationStorage` annotation (see *Section 13.1.1, “Annotation based configuration”*)
- the API for programmatic configuration (see *Section 13.1.2, “Programmatic configuration”*)
- specifying a gloabl default strategy via the `hibernate.ogm.datastore.document.association_storage` configuration property

13.2.3.1. In Entity strategy

With this strategy, Hibernate OGM directly stores the id(s) of the other side of the association into a field or an embedded document depending if the mapping concerns a single object or a collection. The field that stores the relationship information is named like the entity property.

Note

When using this strategy the annotations `@JoinTable` will be ignored because no collection is created for associations.

You can use `@JoinColumn` to change the name of the field that stores the foreign key (as an example, see ???).

Example 13.14. Java entity

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    @ManyToMany
    public Set<BankAccount> bankAccounts;

    // getters, setters, ...
```

Example 13.15. JSON representation

```
{
    "_id": "AccountOwner:id_:owner0001_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "AccountOwner",
    "bankAccounts" : [
        "accountABC",
        "accountXYZ"
    ]
}
```

Example 13.16. Unidirectional one-to-one

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    private Vehicule vehicule;

    // getters, setters ...
}
```

```
{
    "_id": "Vehicule:id_:V001_",
    "_rev": "1-41dc2d2fd68ce2fc683241a60e59a676",
    "$type": "entity",
    "$table": "Vehicule",
    "id": "V001",
    "brand": "Mercedes",
}
```

```
{
  "_id": "Wheel:id_:W1_",
  "_rev": "1-30430d67174484f6b647480dbf781f55",
  "$type": "entity",
  "$table": "Wheel",
  "id": "W1",
  "diameter" : 0,
  "vehicule_id" : "V001"
}
```

Example 13.17. Unidirectional one-to-one with @JoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}


@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @JoinColumn( name = "part_of" )
    private Vehicule vehicule;

    // getters, setters ...
}
```

```
{
    "_id": "Vehicule:id_:V001_",
    "_rev": "1-41dc2d2fd68ce2fc683241a60e59a676",
    "$type": "entity",
    "$table": "Vehicule",
    "id": "V001",
    "brand": "Mercedes",
}
```

```
{
  "_id": "Wheel:id_:W1_",
  "_rev": "1-30430d67174484f6b647480dbf781f55",
  "$type": "entity",
  "$table": "Wheel",
  "id": "W1",
  "diameter" : 0,
  "part_of" : "V001"
}
```

In a true one-to-one association, it is possible to share the same id between the two entities and therefore a foreign key is not required. You can see how to map this type of association in the following example:

Example 13.18. Unidirectional one-to-one with @MapsId and @PrimaryKeyJoinColumn

```
@Entity
public class Vehicule {

    @Id
    private String id;
    private String brand;

    // getters, setters ...
}

@Entity
public class Wheel {

    @Id
    private String id;
    private double diameter;

    @OneToOne
    @PrimaryKeyJoinColumn
    @MapsId
    private Vehicule vehicule;

    // getters, setters ...
}
```

```
{
    "_id": "Vehicule:id_:V001_",
    "_rev": "1-41dc2d2fd68ce2fc683241a60e59a676",
    "$type": "entity",
    "$table": "Vehicule",
    "id": "V001",
    "brand": "Mercedes",
}
```

```
{
  "_id": "Wheel:vehicule/_id_:V001_",
  "_rev": "1-30430d67174484f6b647480dbf781f55",
  "$type": "entity",
  "$table": "Wheel",
  "diameter" : 0,
  "vehicule_id" : "V001"
}
```

Example 13.19. Bidirectional one-to-one

```
@Entity
public class Husband {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Wife wife;

    // getters, setters ...
}

@Entity
public class Wife {

    @Id
    private String id;
    private String name;

    @OneToOne
    private Husband husband;

    // getters, setters ...
}
```

```
{
  "_id": "Husband:id_:alex_",
  "_rev": "2-8f976fc216130fb40144b000910b9c1d",
  "$type": "entity",
  "$table": "Husband",
  "id" : "alex",
  "name" : "Alex",
  "wife" : "bea"
}
```

```
{
  "_id": "Wife:id_:bea_",
  "_rev": "2-69130cc082958becbdf4154a3d19c2e6",
  "$type": "entity",
  "$table": "Wife",
  "id" : "bea",
  "name" : "Bea",
  "husband" : "alex"
}
```

Example 13.20. Unidirectional one-to-many

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id": "Basket:id_:davide/_basket_",
  "_rev": "2-8f976fc216130fb40144b000910b9c1d",
  "$type": "entity",
  "$table": "Basket",
  "id" : "davide_basket",
  "owner" : "Davide",
  "products" : [ "Beer", "Pretzel" ]
}
```

Product collection

```
{
  "_id": "Product:name_:Beer_",
  "_rev": "1-e2a51de970f3e5a0e1118989eef1cf7b",
  "$type": "entity",
  "$table": "Product",
  "name" : "Beer",
  "description" : "Tactical nuclear penguin"
}
{
  "_id": "Product:name_:Pretzel_",
  "_rev": "1-b78ce2687db2fb550d9e8753423db3f3",
  "$type": "entity",
  "$table": "Product",
  "name" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
```

Example 13.21. Unidirectional one-to-many using one collection per strategy with @OrderColumn

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

Basket collection

```
{
  "_id" : "davide_basket",
  "owner" : "Davide"
}
```

Product collection

```
{
  "_id" : "Pretzel",
  "description" : "Glutino Pretzel Sticks"
}
{
  "_id" : "Beer",
  "description" : "Tactical nuclear penguin"
}
```

associations_Basket_Product collection

```
{
  "_id" : { "Basket_id" : "davide_basket" },
  "rows" : [
    {
      "products_name" : "Pretzel",
      "products_ORDER" : 1
    },
    {
      "products_name" : "Beer",
      "products_ORDER" : 0
    }
  ]
}
```

A map can be used to represents an association, in this case Hibernate OGM will store the key of the map and the associated id.

Example 13.22. Unidirectional one-to-many using maps with defaults

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {

    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

```
{
  "_id": "User:id_:user/_001",
  "_rev": "3-77de96250380a79a20a38e78826bf4f7",
  "$type": "entity",
  "$table": "User",
  "id" : "user_001",
  "addresses" : [
    {
      "addresses_KEY" : "work",
      "addresses_id" : "address_001"
    },
    {
      "addresses_KEY" : "home",
      "addresses_id" : "address_002"
    }
  ]
}
```

```
{
  "_id": "Address:id_:address/_001",
  "_rev": "1-dd366cd017f87548956dc55d3b12fefd",
  "$type": "entity",
  "$table": "Address",
  "id" : "address_001",
  "city" : "Rome"
}
```

```
{
  "_id": "Address:id_:address/_001",
  "_rev": "1-04f13666a62473ac951dd039c7cdc780",
  "$type": "entity",
  "$table": "Address",
  "id" : "address_002",
  "city" : "Paris"
}
```

You can use @MapKeyColumn to rename the column containing the key of the map.

Example 13.23. Unidirectional one-to-many using maps with @MapKeyColumn

```
@Entity
public class User {

    @Id
    private String id;

    @OneToMany
    @MapKeyColumn(name = "addressType")
    private Map<String, Address> addresses = new HashMap<String, Address>();

    // getters, setters ...
}

@Entity
public class Address {
    @Id
    private String id;
    private String city;

    // getters, setters ...
}
```

```
{
  "_id": "User:id_:user/_001",
  "_rev": "3-77de96250380a79a20a38e78826bf4f7",
  "$type": "entity",
  "$table": "User",
  "id" : "user_001",
  "addresses" : [
    {
      "addressType" : "work",
      "addresses_id" : "address_001"
    },
    {
      "addressType" : "home",
      "addresses_id" : "address_002"
    }
  ]
}
```

```
{
  "_id": "Address:id_:address/_001",
  "_rev": "1-dd366cd017f87548956dc55d3b12fefd",
  "$type": "entity",
  "$table": "Address",
  "id" : "address_001",
  "city" : "Rome"
}
```

```
{
  "_id": "Address:id_:address/_001",
  "_rev": "1-04f13666a62473ac951dd039c7cdc780",
  "$type": "entity",
  "$table": "Address",
  "id" : "address_002",
  "city" : "Paris"
}
```

Example 13.24. Unidirectional many-to-one

```
@Entity
public class JavaUserGroup {

    @Id
    private String jugId;
    private String name;

    // getters, setters ...
}

@Entity
public class Member {

    @Id
    private String id;
    private String name;

    @ManyToOne
    private JavaUserGroup memberOf;

    // getters, setters ...
}
```

```
{
  "_id": "JavaUserGroups:id_:summer/_camp",
  "_rev": "1-04f13666a62473ac951dd039c7cdc780",
  "$type": "entity",
  "$table": "JavaUserGroup",
  "id" : "summer_camp",
  "name" : "JUG Summer Camp"
}
```

```
{
  "_id": "Member:id_:jerome",
  "_rev": "1-880bf595c39a965dec0216d9d990ebd1",
  "$type": "entity",
  "$table": "Member",
  "id" : "jerome",
  "name" : "Jerome"
  "memberOf_jugId" : "summer_camp"
}
```

```
{
  "_id": "Member:id_:emmanuel",
  "_rev": "1-18e83ce9774a769814c401c49a5afcf3",
  "$type": "entity",
  "$table": "Member",
  "id" : "emmanuel",
  "name" : "Emmanuel Bernard"
  "memberOf_jugId" : "summer_camp"
}
```

Example 13.25. Bidirectional many-to-one

```
@Entity
public class SalesForce {

    @Id
    private String id;
    private String corporation;

    @OneToMany(mappedBy = "salesForce")
    private Set<SalesGuy> salesGuys = new HashSet<SalesGuy>();

    // getters, setters ...
}

@Entity
public class SalesGuy {

    private String id;
    private String name;

    @ManyToOne
    private SalesForce salesForce;

    // getters, setters ...
}
```

```
{
  "_id": "SalesForce:id_:red/_hat",
  "_rev": "1-04f13666a62473ac951dd039c7cdc780",
  "$type": "entity",
  "$table": "SalesForce",
  "_id": "red_hat",
  "corporation": "Red Hat",
  "salesGuys": [ "eric", "simon" ]
}
```

```
{
  "_id": "SalesGuy:id_:eric",
  "_rev": "1-18e83ce9774a769814c401c49a5afcf3",
  "$type": "entity",
  "$table": "SalesGuy",
  "id": "eric",
  "name": "Eric"
  "salesForce_id": "red_hat",
}
```

```
{
  "_id": "SalesGuy:id_:eric",
  "_rev": "1-18e83ce9774a769814c401c49a5afcf3",
  "$type": "entity",
  "$table": "SalesGuy",
  "id": "simon",
  "name": "Simon",
  "salesForce_id": "red_hat"
}
```

Example 13.26. Unidirectional many-to-many using in entity strategy

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private Long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

```
{
   "_id": "ClassRoom:id_:1_",
   "_rev": "2-ae1d9748a84af991615fa842a7e796ea",
   "$type": "entity",
   "$table": "ClassRoom",
   "id": "1",
   "students": [
       "mario",
       "john"
   ],
   "name": "Math"
}
```

```
{
   "_id": "ClassRoom:id_:2_",
   "_rev": "2-0e58f03f518c5c1982bb7936308604e4",
   "$type": "entity",
   "$table": "ClassRoom",
   "id": "2",
   "students": [
       "kate",
       "mario"
   ],
   "name": "English"
}
```

```
{
   "_id": "Student:id_:john_",
   "_rev": "1-60b642619f0e62e079da8a6521ea9750",
   "$type": "entity",
   "$table": "Student",
   "id": "john",
   "name": "John Doe"
}
```

```
{
   "_id": "Student:id_:kate_",
   "_rev": "1-911bb5cbc9b16c6d90f1e91e856a9224",
   "$type": "entity",
   "$table": "Student",
   "id": "kate",
   "name": "Kate Doe"
}
```

```
{
   "_id": "Student:id_:mario_",
   "_rev": "1-7dc611e3c627a837033e7eb5e244f7f8",
   "$type": "entity",
   "$table": "Student",
   "id": "mario",
   "name": "Mario Rossi"
}
```

Example 13.27. Bidirectional many-to-many

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}
```

```
@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany( mappedBy = "bankAccounts" )
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

```
{
   "_id": "AccountOwner:id_:owner/_1_",
   "_rev": "3-07eb9959eac966afedd0547aa74a59a7",
   "$type": "entity",
   "$table": "AccountOwner",
   "id": "owner_1",
   "SSN": "0123456",
   "bankAccounts": [
       "account_1",
       "account_2"
   ]
}
```

```
{
   "_id": "BankAccount:id_:account/_1_",
   "_rev": "2-87252fffa4ab443485f55504215fbed3",
   "$type": "entity",
   "$table": "BankAccount",
   "id": "account_1",
   "accountNumber": "X2345000",
   "owners": [
       "owner_1"
   ]
}
```

```
{
   "_id": "BankAccount:id_:account/_2_",
   "_rev": "2-15bdfeda927dd10fa10aa19ceee4ea34",
   "$type": "entity",
   "$table": "BankAccount",
   "id": "account_2",
   "accountNumber": "ZZZ-009",
   "owners": [
       "owner_1"
   ]
}
```

13.2.3.2. Association document strategy

With this strategy, Hibernate OGM uses separate association documents (with `$type` set to "association") to store all navigation information. Each assocation document is structured in 2 parts. The first is the `_id` field which contains the identifier information of the association owner and the name of the association table. The second part is the `rows` field which stores (into an embedded collection) all ids that the current instance is related to.

Example 13.28. Unidirectional relationship

```
{
   "_id": "AccountOwner_BankAccount:owners/_id_:4f5b48ad-f074-4a64-8cf4-1f9c54a33f76_",
   "_rev": "1-18ef25ec73c1942c45c868aa92f24f2c",
   "$type": "association",
   "rows": [
        7873a2a7-c77c-447c-b000-890f0a4dfa9a
   ]
}
```

For a bidirectional relationship, another document is created where ids are reversed. Don’t worry, Hibernate OGM takes care of keeping them in sync:

Example 13.29. Bidirectional relationship

```
{
   "_id": "AccountOwner_BankAccount:owners/_id_:4f5b48ad-f074-4a64-8cf4-1f9c54a33f76_",
   "_rev": "1-18ef25ec73c1942c45c868aa92f24f2c",
   "$type": "association",
   "rows": [
        "7873a2a7-c77c-447c-b000-890f0a4dfa9a"
   ]
}
{
   "_id": "AccountOwner_BankAccount:bankAccounts/_id_:7873a2a7-c77c-447c-b000-890f0a4dfa9a_",
   "_rev": "1-78e92f980745941a779abb914da65a6c",
   "$type": "association",
   "rows": [
        "4f5b48ad-f074-4a64-8cf4-1f9c54a33f76"
   ]
}
```

Note

This strategy won’t affect *-to-one associations or embedded collections.

Example 13.30. Unidirectional one-to-many using document strategy

```java
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

```json
{
   "_id": "Basket:id_:davide/_basket_",
   "_rev": "1-ba920ac3d1ed5544a71d6c6c5f2ee286",
   "$type": "entity",
   "$table": "Basket",
   "id": "davide_basket",
   "owner": "Davide"
}
```

```json
{
   "_id": "Basket:id_:davide/_basket_",
   "_rev": "1-ba920ac3d1ed5544a71d6c6c5f2ee286",
   "$type": "entity",
   "$table": "Basket",
   "id": "davide_basket",
   "owner": "Davide"
}
```

```json
{
   "_id": "Product:name_:Pretzel_",
   "_rev": "1-b78ce2687db2fb550d9e8753423db3f3",
   "$type": "entity",
   "$table": "Product",
   "description": "Glutino Pretzel Sticks",
   "name": "Pretzel"
}
```

```
{
   "_id": "Basket_Product:Basket/_id_:davide/_basket_",
   "_rev": "1-f6d9aa44a7ca4f01b68c94b1f5599956",
   "$type": "association",
   "rows": [
       "Beer",
       "Pretzel"
   ]
}
```

Using the annotation `@JoinTable` it is possible to change the value of the document containing the association.

Example 13.31. Unidirectional one-to-many using document strategy with `@JoinTable`

```
@Entity
public class Basket {

    @Id
    private String id;

    private String owner;

    @OneToMany
    @JoinTable( name = "BasketContent" )
    private List<Product> products = new ArrayList<Product>();

    // getters, setters ...
}

@Entity
public class Product {

    @Id
    private String name;

    private String description;

    // getters, setters ...
}
```

```
{
   "_id": "Basket:id_:davide/_basket_",
   "_rev": "1-ba920ac3d1ed5544a71d6c6c5f2ee286",
   "$type": "entity",
   "$table": "Basket",
   "id": "davide_basket",
   "owner": "Davide"
}
```

```
{
    "_id": "Basket:id_:davide/_basket_",
    "_rev": "1-ba920ac3d1ed5544a71d6c6c5f2ee286",
    "$type": "entity",
    "$table": "Basket",
    "id": "davide_basket",
    "owner": "Davide"
}
```

```
{
    "_id": "Product:name_:Pretzel_",
    "_rev": "1-b78ce2687db2fb550d9e8753423db3f3",
    "$type": "entity",
    "$table": "Product",
    "description": "Glutino Pretzel Sticks",
    "name": "Pretzel"
}
```

```
{
    "_id": "BasketContent:Basket/_id_:davide/_basket_",
    "_rev": "1-f6d9aa44a7ca4f01b68c94b1f5599956",
    "$type": "association",
    "rows": [
        "Beer",
        "Pretzel"
    ]
}
```

Example 13.32. Unidirectional many-to-many using document strategy

```
@Entity
public class Student {

    @Id
    private String id;
    private String name;

    // getters, setters ...
}

@Entity
public class ClassRoom {

    @Id
    private Long id;
    private String lesson;

    @ManyToMany
    private List<Student> students = new ArrayList<Student>();

    // getters, setters ...
}
```

```
{
   "_id": "ClassRoom:id_:1_",
   "_rev": "2-ae1d9748a84af991615fa842a7e796ea",
   "$type": "entity",
   "$table": "ClassRoom",
   "id": "1",
   "name": "Math"
}
```

```
{
   "_id": "ClassRoom:id_:2_",
   "_rev": "2-0e58f03f518c5c1982bb7936308604e4",
   "$type": "entity",
   "$table": "ClassRoom",
   "id": "2",
   "name": "English"
}
```

```
{
   "_id": "Student:id_:john_",
   "_rev": "1-60b642619f0e62e079da8a6521ea9750",
   "$type": "entity",
   "$table": "Student",
   "id": "john",
   "name": "John Doe"
}
```

```
{
   "_id": "Student:id_:kate_",
   "_rev": "1-911bb5cbc9b16c6d90f1e91e856a9224",
   "$type": "entity",
   "$table": "Student",
   "id": "kate",
   "name": "Kate Doe"
}
```

```
{
   "_id": "Student:id_:mario_",
   "_rev": "1-7dc611e3c627a837033e7eb5e244f7f8",
   "$type": "entity",
   "$table": "Student",
   "id": "mario",
   "name": "Mario Rossi"
}
```

```
{
   "_id": "ClassRoom_Student:ClassRoom/_id_:1_",
   "_rev": "1-351e470a8c134a084d9ad282796a7464",
   "$type": "association",
   "rows": [
       "mario",
       "john"
   ]
}
```

```
{
   "_id": "ClassRoom_Student:ClassRoom/_id_:2_",
   "_rev": "1-825d1900ec216dc73e0152564de8e975",
   "$type": "association",
   "rows": [
       "kate"
   ]
}
```

Example 13.33. Bidirectional many-to-many using document strategy

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    private String SSN;

    @ManyToMany
    private Set<BankAccount> bankAccounts;

    // getters, setters ...
}

@Entity
public class BankAccount {

    @Id
    private String id;

    private String accountNumber;

    @ManyToMany(mappedBy = "bankAccounts")
    private Set<AccountOwner> owners = new HashSet<AccountOwner>();

    // getters, setters ...
}
```

```
{
   "_id": "AccountOwner:id_:owner/_1_",
   "_rev": "3-07eb9959eac966afedd0547aa74a59a7",
   "$type": "entity",
   "$table": "AccountOwner",
   "id": "owner_1",
   "SSN": "0123456",
}
```

```
{
   "_id": "BankAccount:id_:account/_1_",
   "_rev": "2-87252fffa4ab443485f55504215fbed3",
   "$type": "entity",
   "$table": "BankAccount",
   "id": "account_1",
   "accountNumber": "X2345000",
}
```

```
{
   "_id": "BankAccount:id_:account/_2_",
   "_rev": "2-15bdfeda927dd10fa10aa19ceee4ea34",
   "$type": "entity",
   "$table": "BankAccount",
   "id": "account_2",
   "accountNumber": "ZZZ-009",
}
```

```
{
   "_id": "AccountOwner_BankAccount:bankAccounts/_id_:account/_1_",
   "_rev": "1-34ecb6bcadae6e51112de0cf50387521",
   "$type": "association",
   "rows": [
       "owner_1"
   ]
}
```

```
{
   "_id": "AccountOwner_BankAccount:bankAccounts/_id_:account/_2_",
   "_rev": "1-34ecb6bcadae6e51112de0cf50387521",
   "$type": "association",
   "rows": [
       "owner_1"
```

```
    ]
}
```

```
{
    "_id": "AccountOwner_BankAccount:owners/_id_:owner/_1_",
    "_rev": "2-d2cc7816eae5498a0829a3cdae0b208e",
    "$type": "association",
    "rows": [
        "account_1",
        "account_2"
    ]
}
```

13.3. Transactions

CouchDB does not support transactions. Only changes applied to the same document are done atomically. A change applied to more than one document will not be applied atomically. This problem is slightly mitigated by the fact that Hibernate OGM queues all changes before applying them during flush time. So the window of time used to write to CouchDB is smaller than what you would have done manually.

We recommend that you still use transaction demarcations with Hibernate OGM to trigger the flush operation transparently (on commit). But do not consider rollback as a possibility, this won’t work.

13.4. Queries

Hibernate OGM is a work in progress and we are actively working on JP-QL query support.

In the mean time, you have two strategies to query entities stored by Hibernate OGM:

- use native CouchDB queries
- use Hibernate Search

Because Hibernate OGM stores data in CouchDB in a natural way, you can the HTTP client or REST library of your choice and execute queries (using CouchDB views) on the datastore directly without involving Hibernate OGM. The benefit of this approach is to use the query capabilities of CouchDB. The drawback is that raw CouchDB documents will be returned and not managed entities.

The alternative approach is to index your entities with Hibernate Search. That way, a set of secondary indexes independent of CouchDB is maintained by Hibernate Search and you can write queries on top of them. The benefit of this approach is an nice integration at the JPA / Hibernate API level (managed entities are returned by the queries). The drawback is that you need to store the Lucene indexes somewhere (file system, infinispan grid etc). Have a look at the Infinispan section for more info on how to use Hibernate Search.