Hibernate OGM Reference Guide

4.0.0-SNAPSHOT

by Emmanuel Bernard (Red Hat), Sanne Grinovero (Red Hat), and Gunnar Morling (Red Hat)

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 still very young and very ambitious at the same time. Many things are on the roadmap (more NoSQL, query, denormalization engine, etc). If you wish to help, please check *Chapter 1, How to get help and contribute on Hibernate OGM*.

Hibernate OGM is released under the LGPL open source license.

Warning

This documentation and this project are work in progress. Please give us feedback on

- what you like
- what you don't like
- what is confusing

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

1. Goals

Hibernate OGM:

- offers a familiar programming paradigm 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

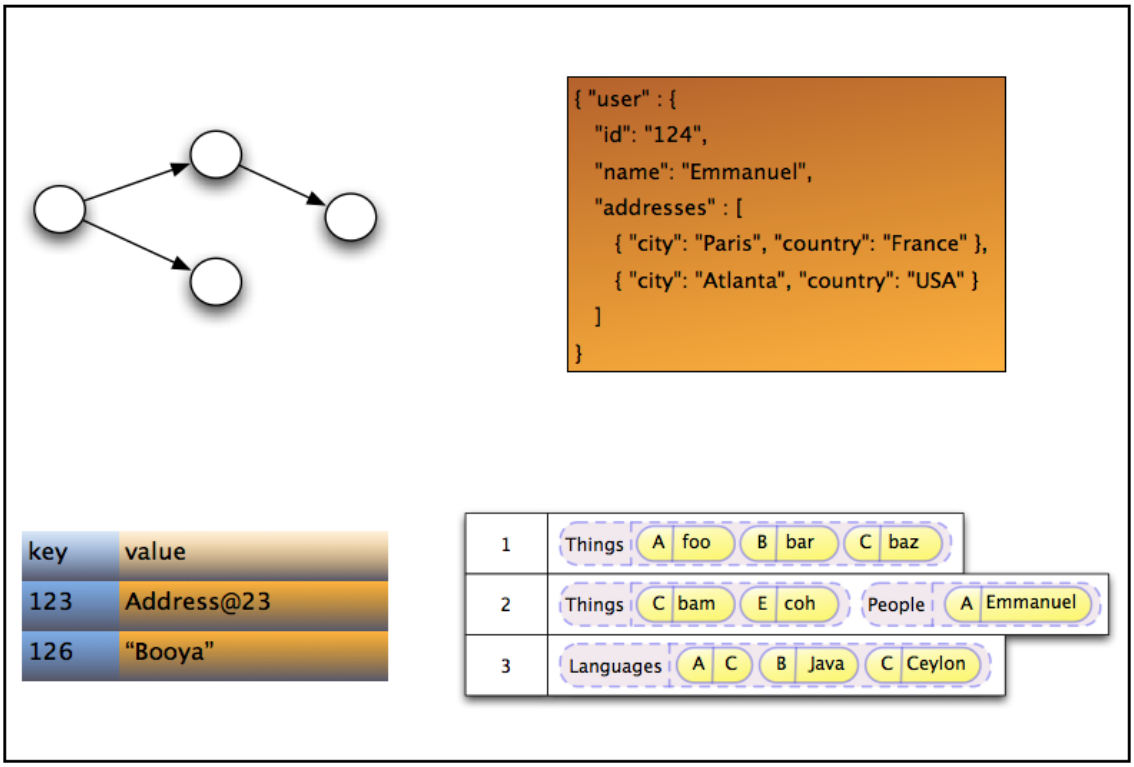


Figure 1. Various NoSQL families

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)
- 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 Hibernate Search's full-text queries
- JPA and native Hibernate ORM API support

3. Use cases

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

- need to scale your data store 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 while using a familiar programming model
- use a familiar JPA front end to datagrids (in particular Infinispan)
- use Hibernate Search full-text search / text analysis capabilities and store the data set in an elastic grid

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. The code, the direction and the documentation are all in flux and being built by the community. 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-infinispan:4.0.0-SNAPSHOT` to your project dependencies.

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan</artifactId>
    <version>4.0.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.0-api</artifactId>
    <version>1.0.1.Final</version>
</dependency>
```

Note

While Hibernate OGM depends on JPA 2.0, 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.

We will use the JPA APIs in this tutorial.

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. The final list of dependencies should look like this:

```
<dependencies>
    <!-- Hibernate OGM dependency -->
    <dependency>
        <groupId>org.hibernate.ogm</groupId>
        <artifactId>hibernate-ogm-core</artifactId>
        <version>4.0.0-SNAPSHOT</version>
    </dependency>

    <!-- standard APIs dependencies - provided in a Java EE container -->
    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.0-api</artifactId>
        <version>1.0.1.Final</version>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.transaction</groupId>
        <artifactId>jboss-transaction-api_1.1_spec</artifactId>
        <version>1.0.0.Final</version>
        <scope>provided</scope>
    </dependency>
```

```
    <!-- JBoss Transactions dependency -->
    <dependency>
        <groupId>org.jboss.jbossts</groupId>
        <artifactId>jbossjta</artifactId>
        <version>4.16.4.Final</version>
    </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 optional if you plan and use Infinispan, otherwise adjust to your favorite
                NoSQL Datastore provider.
            <property name="hibernate.ogm.datastore.provider"

 value="org.hibernate.ogm.datastore.infinispan.impl.InfinispanDatastoreProvider"/>
            -->
            <!-- 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.

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

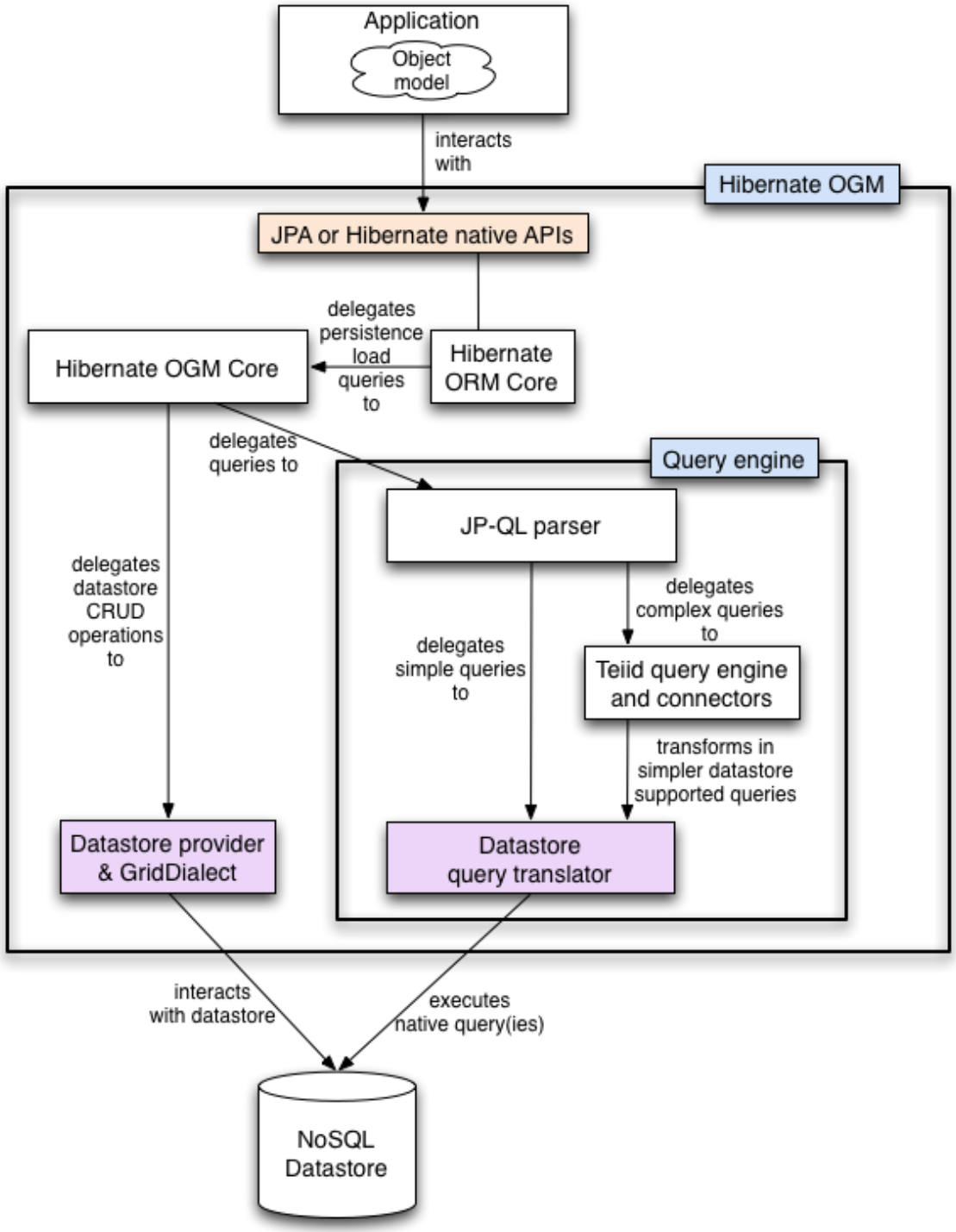


Figure 3.1. General architecture

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 four 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 query is too complex for the native capabilities of the NoSQL store, the Teiid query engine is used as an intermediary engine to implement the missing features (typically joins between entities, aggregation). Finally, if the underlying engine does not have any query support, we use Hibernate Search as an external query engine.

Reality is a bit more nuanced, 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 including:

- 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 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)

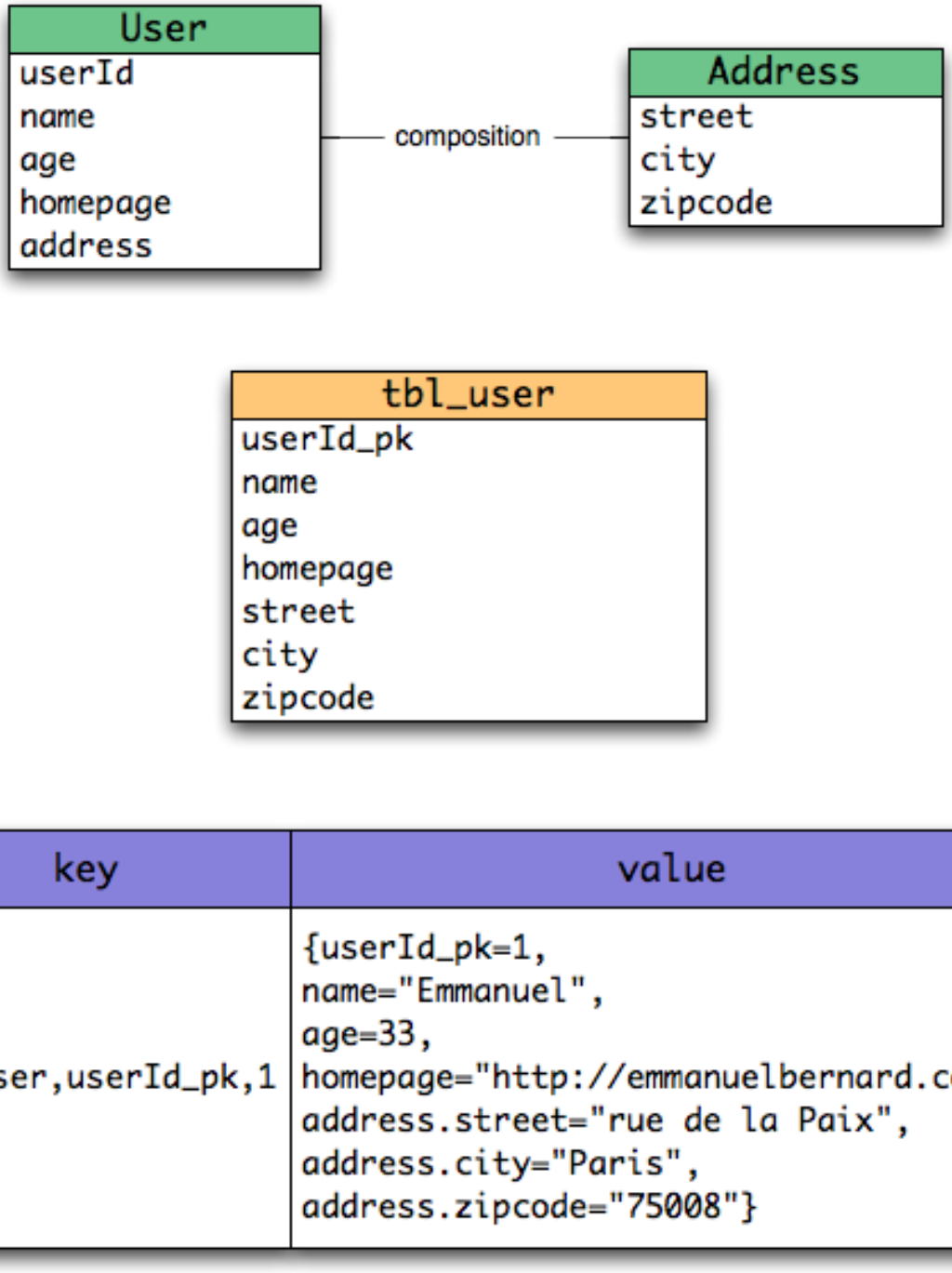


Figure 3.2. Storing entities

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.

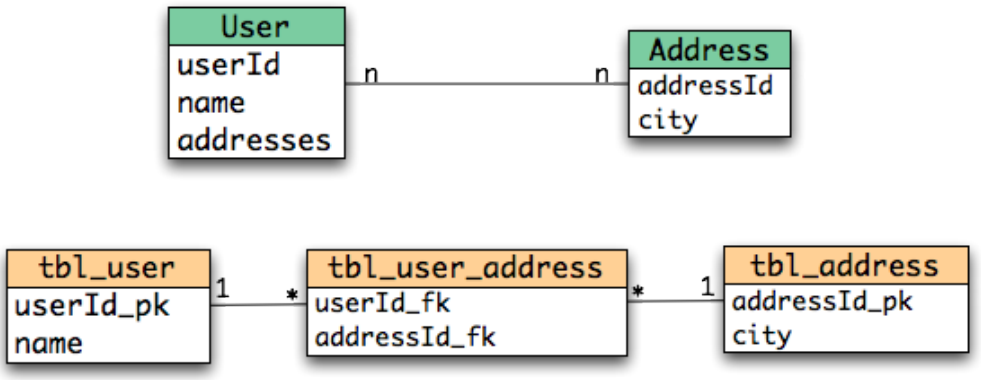
- for a document oriented store, the map is represented by a document and each entry in the map corresponds to a property in a document.

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

Note

We might offer alternative association data persistence options in the future based on feedback.

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.

TODO: this sentence might be worth a diagram to show the difference with the key/value 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

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

Note

Query support is in active development. This section describes where the project is going.

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 Teiid as an intermediary engine, Teiid delegating parts of the query 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 identifiers snd uses Hibernate OGM to return managed objects.

Some of the JP-QL features are not supported by NoSQL solutions. Two typical examples are joins between entities - which you should limit anyways in a NoSQL environment - and aggregations like average, max, min etc. When the NoSQL store does not support the query, we use Teiid - a database federation engine - to build simpler queries executed to the datastore and perform the join or aggregation operations in Teiid itself.

Finally 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:

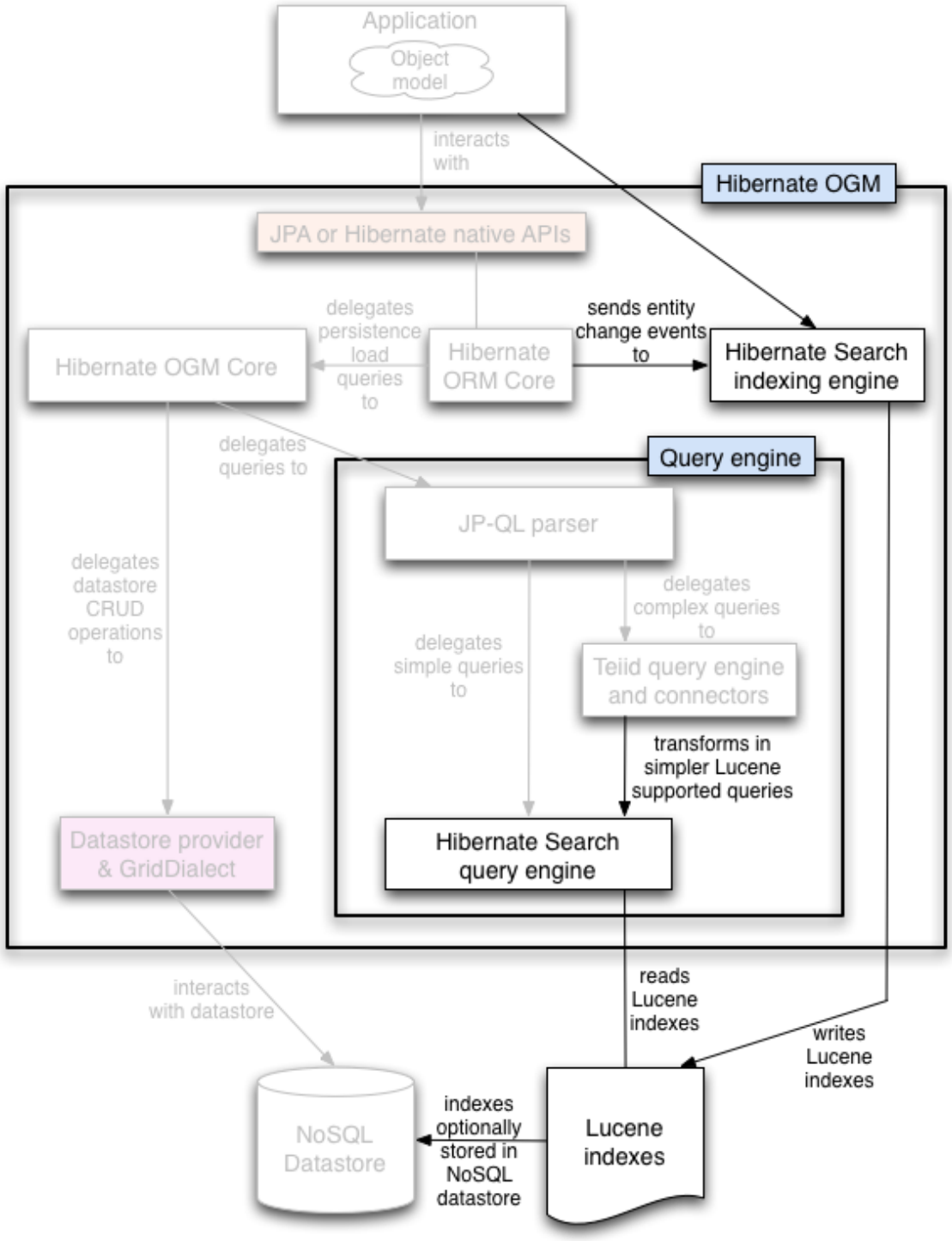


Figure 3.4. Using Hibernate Search as query engine - greyed areas are blocks already present in Hibernate OGM's architecture

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.

Note that for complex queries involving joins or aggregation, Hibernate OGM can use Teiid as an intermediary query engine that will delegate to Hibernate Search.

Note that 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)

What's the progress status on queries?

Well… now is a good time to remind you that Hibernate OGM is open source and that contributing to such cutting edge project is a lot of fun. Check out *Chapter 1, How to get help and contribute on Hibernate OGM* for more details.

But to answer your question, we have finished the skeleton of the architecture as well as the JP-QL parser implementation. The Hibernate Search query translator can execute simple queries already. However, we do not yet have a NoSQL specific query translator but the approach is quite clear to us. Teiid for complex queries is also not integrated but work is being done to facilitate that integration soon. Native Hibernate Search queries are fully supported.

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 5, 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 5, 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 JBoss AS, 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 JBoss AS 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 5, 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.

In other words, 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 5.1.5, “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 JBoss AS 7.2

Provided you’re deploying on JBoss AS 7.2 or JBoss EAP6, there is an additional way to add the OGM dependencies to your application.

In JBoss AS 7, 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 AS 7.2* [https://docs.jboss.org/author/display/AS72/Class+Loading+in+AS7].

You can download the pre-packaged module from:

- *Sourceforge* [https://downloads.sourceforge.net/project/hibernate/hibernate-ogm/4.0.0-SNAPSHOT/hibernate-ogm-modules-4.0.0-SNAPSHOT-jbossas-72-dist.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&v=4.0.0-SNAPSHOT&e=zip&c=jbossas-72-dist]

Unpack the archive into the `modules` folder of your JBoss AS 7.2 installation. The modules included are:

- *org.hibernate:ogm*, containing the core OGM library and the infinispan datastore provider.
- *org.hibernate.ogm.ehcache:main*, containing the ehcache datastore provider.
- *org.hibernate.ogm.mongodb:main*, containing the mongodb datastore provider.
- *org.hibernate:main*, containing the latest hibernate ORM libraries compatible with OGM.

Warning

The `org.hibernate:main` module changes the version of Hibernate ORM included in the default JBoss AS 7.2.

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

Using the manifest
: Add this entry to the MANIFEST.MF in your archive:

```
Dependencies: org.hibernate:ogm services
```

Using jboss-deployment-structure.xml
: This is a proprietary JBoss AS descriptor. Add a `WEB-INF/jboss-deployment-structure.xml` in your archive with content:

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

```
</jboss-deployment-structure>
```

More information about the descriptor can be found in the *JBoss AS 7.2 documentation* [https://docs.jboss.org/author/display/AS72/Class+Loading+in+AS7].

Datastores

Currently Hibernate OGM supports the following 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)

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

Hibernate OGM interacts with NoSQL datastores via two contracts:

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

The main thing you need to do is to configure which datastore provider you want to use. This is done via the `hibernate.ogm.datastore.provider` option. Possible values are

- the fully qualified class name of a `DatastoreProvider` implementation or
- one preferably of the following shortcuts: `map` (only to be used for unit tests), `infinispan`, `ehcache`, `mongodb`, `neo4j` or `couchdb`

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, MyDatastoreProvider.class );
// or an instance
properties.put( OgmProperties.DATASTORE_PROVIDER, new MyDatastoreProvider() );

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

You also need to add the relevant Hibernate OGM module in your classpath. In maven that would look like:

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan</artifactId>
    <version>4.0.0-SNAPSHOT</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.

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 happy.

5.1. 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'll 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].

5.1.1. Configure Infinispan

Two steps basically:

- Add the dependencies to classpath
- And then choose one of:

- Use the default Infinispan configuration (no action needed)
- Point to your own configuration resource name
- Point to a `JNDI` name of an existing Infinispan instance

5.1.1.1. Adding Infinispan dependencies

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

```
<dependency>
    <groupId>org.hibernate.ogm</groupId>
    <artifactId>hibernate-ogm-infinispan</artifactId>
    <version>4.0.0-SNAPSHOT</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`

5.1.1.2. 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.

Infinispan datastore configuration properties

`hibernate.ogm.datastore.provider`
: To use Infinispan as a datastore provider set it to `infinispan`.

`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.

`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`.

Note

When bootstrapping a session factory or entity manager factory programmatically, you should use the constants accessible via `InfinspanProperties` when specifying the configuration properties listed above. Common properties shared between stores are declared on `OgmProperties`. To ease migration between stores, it is recommended to reference these constants directly from there.

5.1.1.3. Cache names used by Hibernate OGM

Hibernate OGM will not use a single Cache but three and is going to use them for different purposes; so that you can configure the Caches meant for each role separately.

Infinispan cache names and purpose

`ENTITIES`
: Is going to be used to store the main attributed of your entities.

`ASSOCIATIONS`
: Stores the association information which maps to the relations between your entities.

`IDENTIFIER_STORE`
: Contains internal metadata that Hibernate OGM needs to provide sequences and auto-incremental numbers for primary key generation.

We'll explain in the following paragraphs how you can take advantage of this and 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* [https://docs.jboss.org/author/display/ISPN/User+Guide] 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 5.1. Simple structure of an infinispan xml configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:infinispan:config:6.0 http://www.infinispan.org/schemas/infinispan-
config-6.0.xsd"
    xmlns="urn:infinispan:config:6.0">


    <global>
```

```
    </global>

    <default>
    </default>

    <namedCache name="ENTITIES">
    </namedCache>

    <namedCache name="ASSOCIATIONS">
    </namedCache>

    <namedCache name="IDENTIFIERS">
    </namedCache>

</infinispan>
```

The `global` section contains elements which affect the whole instance; mainly of interest for Hibernate OGM users is the `transport` element in which we'll set JGroups configuration overrides.

In the `namedCache` section (or in `default` if we want to affect all named caches) we'll likely want to configure clustering modes, eviction policies and `CacheStore`s.

5.1.2. 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 a trivial programming exercise.

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 5.2. Enabling a FileCacheStore and eviction

```
<namedCache name="ENTITIES">
  <eviction strategy="LIRS" maxEntries="2000" />
  <loaders
    passivation="true" shared="false">
    <loader
      class="org.infinispan.loaders.file.FileCacheStore"
      fetchPersistentState="false"
      purgeOnStartup="false">
      <properties>
        <property name="location" value="/var/hibernate-ogm/myapp/entities-data" />
      </properties>
    </loader>
  </loaders>
</namedCache>
```

In this example we enabled both `eviction` and a `CacheStore` (the `loader` 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.

Tip

Currently with Infinispan 5.1, the `FileCacheStore` is neither very fast nor very efficient: we picked it for ease of setup. For a production system it's worth looking at the large collection of high performance and cloud friendly cachestores provided by the Infinispan distribution.

5.1.3. Clustering: deploy 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 server, 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 5.3. 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 5.4. 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:6.0 http://www.infinispan.org/schemas/infinispan-
config-6.0.xsd"
    xmlns="urn:infinispan:config:6.0">

    <global>
        <transport
            clusterName="HibernateOGM-Infinispan-cluster">
            <properties>
                <property name="configurationFile" value="my-jgroups-conf.xml" />
            </properties>
        </transport>
    </global>

    <default>
        <clustering
            mode="distribution" />
    </default>

    <!-- Cache to store the OGM entities -->
    <namedCache
        name="ENTITIES">
    </namedCache>

    <!-- Cache to store the relations across entities -->
    <namedCache
        name="ASSOCIATIONS">
    </namedCache>

    <!-- Cache to store identifiers -->
    <namedCache
        name="IDENTIFIERS">
        <!-- Override the cache mode: -->
        <clustering
            mode="replication" />
    </namedCache>

</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 `ENTITIES` and the `ASSOCIATIONS` caches. But for `IDENTIFIERS` 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.

5.1.4. 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 5.5. Configuring a JBoss Standalone TransactionManager lookup

```
<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* [https://docs.jboss.org/author/display/ISPN/User+Guide] for more advanced configuration options.

5.1.5. 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 filesystems 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 5.1.2, "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:4.5.0.CR1` 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 5.1. 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.

5.2. Ehcache

When combined with Hibernate ORM, Ehcache is commonly used as a 2nd level cache, so caching data which is stored in 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].

5.2.1. 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

5.2.1.1. 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>4.0.0-SNAPSHOT</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`

5.2.1.2. 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`.

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`. To ease migration between stores, it is recommended to reference these constants directly from there.

5.2.2. 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].

5.3. MongoDB

MongoDB [http://www.mongodb.org] is a document oriented datastore written in C++ with strong emphasis on ease of use.

5.3.1. Configuring MongoDB

This implementation is based upon the MongoDB Java driver. The currently supported version is 2.10.1.

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 5.3.1.2, "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.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.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) 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.

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.AssociationDocumentType` 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 `com.mongodb.WriteConcern` enum): `ERRORS_IGNORED`, `ACKNOWLEDGED`, `UNACKNOWLEDGED`, `FSYNCED`, `JOURNALED`, `NONE`, `NORMAL`, `SAFE`, `MAJORITY`, `FSYNC_SAFE`, `JOURNAL_SAFE`, `REPLICAS_SAFE`. For more information, please refer to the *official documentation* [http://api.mongodb.org/java/current/com/mongodb/WriteConcern.html]. This option is case insensitive and the default value is `ACKNOWLEDGED`.

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 (document) stores are declared on `OgmProperties` and `DocumentStoreProperties`, respectively. To ease migration between stores, it is recommended to reference these constants directly from there.

5.3.1.1. Annotation based configuration

Hibernate OGM allows to configure store-specific options via Java annotations. When working with the MongoDB backend, you can specify how associations should be stored using the `AssociationStorage` and `AssociationDocumentStorage` annotations (refer to *Section 5.3.2, "Storage principles"* to learn more about these options).

The following shows an example:

Example 5.6. Configuring the association storage strategy using annotations

```
@Entity
@AssociationStorage(AssociationStorageType.ASSOCIATION_DOCUMENT)
@AssociationDocumentStorage(AssociationDocumentType.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;

    //...
}
```

The annotations on the entity level express 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.

5.3.1.2. 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 MongoDB, you can currently configure the following options using the API:

- association storage strategy (on the global, entity and property level)
- association document storage strategy (on the global, entity and property level)

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

Example 5.7. Example of an option configurator

```
public class MyOptionConfigurator extends OptionConfigurator {
```

```
    @Override
    public void configure(Configurable configurable) {
        configurable.configureOptionsFor( MongoDB.class )
            .entity( Zoo.class )
                .associationStorage( AssociationStorageType.ASSOCIATION_DOCUMENT )
              .associationDocumentStorage( AssociationDocumentType.COLLECTION_PER_ASSOCIATION )
                .property( "animals", ElementType.FIELD )
                    .associationStorage( AssociationStorageType.IN_ENTITY )
            .entity( Animal.class )
                .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 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 `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.

5.3.2. 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.

5.3.2.1. Entities

Entities are stored as MongoDB 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.

Note that embedded objects are mapped as nested documents.

Example 5.8. Example of an entity with an embedded object

```
@Entity
public class News {
    @Id
    private String id;
    private String title;
    @Column(name="desc")
    private String description;
    @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",
    "desc": "This paper discuss why NoSQL will save the world for good",
    "paper": {
        "name": "NoSQL journal of prophecies",
        "owner": "Delphy"
    }
}
```

5.3.2.1.1. Identifiers

The `_id` field of a MongoDB document is directly used to store the identifier columns mapped in the entities. That means you can use simple identifiers (no matter the Java type used) as well as Embedded identifiers. Embedded identifiers are stored as embedded document into the `_id` field. 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` (the recommended approach in MongoDB). That means in particular that MongoDB will automatically index your _id fields. Let's look at an example:

Example 5.9. Example of an entity using Embedded id

```
@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" :{
        "title": "How does Hibernate OGM MongoDB work?",
        "author": "Guillaume"
    }
}
```

5.3.2.2. Associations

Hibernate OGM MongoDB proposes three strategies to store navigation information for associations. To switch between these strategies, either use the `@AssocationStorage` and `@AssociationDocumentStorage` annotations (see *Section 5.3.1.1, "Annotation based configuration"*), the API for programmatic configuration (see *Section 5.3.1.2, "Programmatic configuration"*) or specify a default strategy via the `hibernate.ogm.datastore.document.association_storage` and `hibernate.ogm.mongodb.association_document_storage` configuration properties.

The three possible strategies are:

- IN_ENTITY (default)
- ASSOCIATION_DOCUMENT, using a global collection for all associations
- ASSOCIATION_DOCUMENT, using a dedicated collection for each association

5.3.2.2.1. In Entity strategy

In 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.

Example 5.10. Java entity

```
@Entity
public class AccountOwner {

    @Id
    private String id;
```

```
    @ManyToMany
    public Set<BankAccount> bankAccounts;

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

Example 5.11. JSON representation

```
{
    "_id" : "owner0001",
    "bankAccounts" : [
        { "bankAccounts_id" : "accountXYZ" }
    ]
}
```

5.3.2.2.2. Global collection strategy

With this strategy, Hibernate OGM creates a single collection in which it will store all navigation information for all associations. Each document of this collection is structure 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 5.12. Unidirectional relationship

```
{
    "_id": {
        "owners_id": "owner0001",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [
        { "bankAccounts_id": "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 5.13. Bidirectional relationship

```
{
    "_id": {
        "owners_id": "owner0001",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [{
        "bankAccounts_id": "accountXYZ"
```

```
        }]
}
{
    "_id": {
        "bankAccounts_id": "accountXYZ",
        "table": "AccountOwner_BankAccount"
    },
    "rows": [{
        "owners_id": "owner0001"
    }]
}
```

5.3.2.2.3. 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`. The prefix is useful to quickly identify the association collections from the entity collections. 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

Example 5.14. Unidirectional relationship

```
{
    "_id" : { "owners_id" : "owner0001" },
    "rows" : [
        { "bankAccounts_id" : "accountXYZ" }
    ]
}
```

Example 5.15. Bidirectional relationship

```
{
    "_id" : { "owners_id" : "owner0001" },
    "rows" : [
        { "bankAccounts_id" : "accountXYZ" }
    ]
}
{
    "_id" : { "bankAccounts_id" : "accountXYZ" },
    "rows" : [
        { "owners_id" : "owner0001" }
    ]
```

```
}
```

5.3.3. 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.

5.3.4. 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)

5.3.4.1. 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`

Queries using these constructs will be transformed into equivalent native MongoDB queries.

5.3.4.2. Native MongoDB queries

Hibernate OGM supports native queries for MongoDB with some limitations:

- parameters are not supported
- the result of a native query must be mapped to one entity

If your use case meets these restrictions you can execute a native query like in the following example:

Example 5.16. Using the JPA API

```
@Entity
public class Poem {

    @Id
    private Long id;

    private String name;

    private String author;

   // getters, setters ...

}

...

javax.persistence.EntityManager em = ...

String query1 = "{ $and: [ { name : 'Portia' }, { author : 'Oscar Wilde' } ] }";
Poem poem = (Poem) em.createNativeQuery( query1, Poem.class ).getSingleResult();

String query2 = "{ $query : { author : 'Oscar Wilde' }, $orderby : { name : 1 } }";
List<Poem> poems = em.createNativeQuery( query2, Poem.class ).getResultList();
```

Example 5.17. Using the Hibernate native API

```
org.hibernate.Session session = ...

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

String query2 = "{ $query : { author : 'Oscar Wilde' }, $orderby : { name : 1 } }";
List<Poem> poems = session.createSQLQuery( query2 )
                    .addEntity( "Poem", Poem.class )
                    .list();
```

Note

The method in `Session#createSQLQuery(…)` might look misleading since we are not running a SQL query but the `Session` API was initially thought for relational databases and we decided it was simpler to reuse the same method than invent something new that could increase the confusion.

The result of the query is a managed entity or a list of managed entities. Just like you would get from a JP-QL query.

Native queries can also be created using the `@NamedNativeQuery` annotation:

Example 5.18. 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.

5.3.4.3. 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 5.1.5, "Storing a Lucene index in Infinispan"*) for more info on how to use Hibernate Search.

5.4. 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.

This is our first version and a bit experimental. In particular we plan on using node navigation much more than index lookup in a future version.

5.4.1. 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>4.0.0-SNAPSHOT</version>
</dependency>
```

Alternatively you can find the required libraries in the distribution package on *SourceForge* [https://downloads.sourceforge.net/project/hibernate/hibernate-ogm/4.0.0-SNAPSHOT/hibernate-ogm-modules-4.0.0-SNAPSHOT-jbossas-72-dist.zip]

2. Add the following properties:

```
hibernate.ogm.datastore.provider = neo4j_embedded
hibernate.ogm.neo4j.database_path = C:\example\mydb
```

5.4.2. 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.ogm.neo4j.index.entity (optional)
: Name of the neo4j index containing the stored entities. Default to `_nodes_ogm_index`

hibernate.ogm.neo4j.index.association (optional)
: Name of the Neo4j index containing the stored associations. Default to `_relationships_ogm_index`

hibernate.ogm.neo4j.index.sequence (optional)
: Name of the index that stores the next available value for a sequence. Default to `_sequences_ogm_index`

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`. To ease migration between stores, it is recommended to reference these constants directly from there.

5.4.3. Storage principles

5.4.3.1. Entities

Entities are stored as Neo4j nodes, which means each entity property will be translated into a property of the node. An additional property is added to the node and it contains the name of the table representing the entity.

Example 5.19. Example of entities and the list of properties contained in the corresponding node

```
@Entity
class Account {

    @Id
    String login;
    String password;
    Address homeAddress;

    //...
}

@Embeddable
class Address {
    String city;
    String zipCode;

  //...
}
```

```
Node properties:
   _table
   id
   login
   password
   homeAddress_city
   homeAddress_zipCode
```

The `_table` property has been added by OGM and it contains the name of the table representing the entity (`Account` in this simple case).

5.4.3.2. Associations

Associations are mapped using Neo4j relationships. A unidirectional association is mapped with a relationship between two nodes that start from the node representing the owner of the association. The name of the association is saved as type of the relationship. A bidirectional association is represented by two relationships, one per direction, between the two nodes.

5.4.4. Transactions

Neo4j operations can be executed only inside a transaction. Unless a different `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform` is specified, OGM will integrate with the Neo4j transaction mechanism, this means that you should start and commit transaction using the hibernate session.

Example 5.20. 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();
```

5.5. CouchDB

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. 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].

5.5.1. 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`

hibernate.ogm.option.configurator
: The fully-qualified class name or an instance of a programmatic option configurator (see *Section 5.5.1.2, "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.

5.5.1.1. 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 5.5.2, "Storage principles"* to learn more about association storage strategies in general).

The following shows an example:

Example 5.21. 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.

5.5.1.2. 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 5.22. 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.

5.5.2. 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.

5.5.2.1. 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.

Note

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 5.23. 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;

    @Column(name="desc")
    private String description;

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

```
{
    "_id": "News:id_:news-1_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "News",
    "title": "On the merits of NoSQL",
    "desc": "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.

Embedded objects are mapped as nested documents. The following listing shows an example:

Example 5.24. Example of an entity with an embedded object

```
@Entity
public class News {

    @Id
    private String id;

    @Version
    @Generated
    @Column(name="_rev")
    private String revision;

    private String title;
```

```
    @Column(name="desc")

    private String description;

    @Embedded
    private NewsPaper paper;

    //getters, setters ...
}

@Embeddable
public class NewsPaper {

    private String name;
    private String owner;

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

```
{
    "_id": "News:id_:news-1_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "News",
    "title": "On the merits of NoSQL",
    "desc": "This paper discuss why NoSQL will save the world for good",
    "paper": {
        "name": "NoSQL journal of prophecies",
        "owner": "Delphy"
    }
}
```

5.5.2.1.1. 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.

5.5.2.2. Associations

Hibernate OGM CouchDB provides two strategies to store navigation information for associations. To switch between these strategies, either use the `@AssocationStorage` annotation (see *Section 5.5.1.1, "Annotation based configuration"*), the API for programmatic configuration (see *Section 5.5.1.2, "Programmatic configuration"*) or specify a global default strategy via the `hibernate.ogm.datastore.document.association_storage` configuration property.

The possible strategies are `IN_ENTITY` (default) and `ASSOCIATION_DOCUMENT`.

5.5.2.2.1. 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.

Example 5.25. Java entity

```
@Entity
public class AccountOwner {

    @Id
    private String id;

    @ManyToMany
    public Set<BankAccount> bankAccounts;

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

Example 5.26. JSON representation

```
{
    "_id": "AccountOwner:id_:owner0001_",
    "_rev": "1-d1cd3b00a677a2e31cd0480a796e8480",
    "$type": "entity",
    "$table": "AccountOwner",
    "bankAccounts" : [
        { "bankAccounts_id" : "accountXYZ" }
    ]
}
```

5.5.2.2.2. 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 structure 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 5.27. Unidirectional relationship

```
{
   "_id": "AccountOwner_BankAccount:owners/_id_:4f5b48ad-f074-4a64-8cf4-1f9c54a33f76_",
   "_rev": "1-18ef25ec73c1942c45c868aa92f24f2c",
   "$type": "association",
   "rows": [
```

```
        {
            "bankAccounts_id": "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 5.28. Bidirectional relationship

```
{
   "_id": "AccountOwner_BankAccount:owners/_id_:4f5b48ad-f074-4a64-8cf4-1f9c54a33f76_",
   "_rev": "1-18ef25ec73c1942c45c868aa92f24f2c",
   "$type": "association",
   "rows": [
       {
           "bankAccounts_id": "7873a2a7-c77c-447c-b000-890f0a4dfa9a"
       }
   ]
}
{
   "_id": "AccountOwner_BankAccount:bankAccounts/_id_:7873a2a7-c77c-447c-b000-890f0a4dfa9a_",
   "_rev": "1-78e92f980745941a779abb914da65a6c",
   "$type": "association",
   "rows": [
       {
           "owners_id": "4f5b48ad-f074-4a64-8cf4-1f9c54a33f76"
       }
   ]
}
```

5.5.3. 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.

5.5.4. 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.

Map your entities

TODO:

- Talk about supported approaches (properties, embedded objects, inheritance)
- Talk about associations
- Talk about identifier recommendations

6.1. Supported entity mapping

This section is a work in progress, if you find something that does not work as expected, let us know and we will update it (and fix the problem of course).

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. f 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"*).

All SQL related constructs as well as HQL centered mapping are not supported in Hibernate OGM. Here is a list of feature that will not work:

- Named queries
- Native queries

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

Example 6.1. Using a UUID generator

```
@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;
}
```

6.2. Supported Types

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

6.2.1. Types mapped as native Java Types

A few types are supported natively (ie serialized as is in the tuple data structure):

- Boolean
- Byte
- Calendar (this may change)
- Class (this may change)
- Date (this may change)
- Double
- Integer
- Long
- Byte Array
- String

Warning

This list is subject to change and specifically be reduced to a smaller set of core types.

6.2.2. Types mapped as Strings

For non basic Java types support, OGM stores the data of the object as a String in the data store. Serialisation to a String value is done with cross platform compatibility in mind when required.

- 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

6.3. 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. Check *Section 3.2, "How is data persisted"* for more information.

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.

Query your entities

To query a NoSQL database is a complex feat, especially as not all NoSQL solutions support all forms of query. One of the goals of Hibernate OGM is to deal with this complexity so that users don't have to. However, that's not yet all implemented and depending on your use case there might be better approaches you can take advantage of.

If you skipped to this section without reading *Chapter 3, Architecture*, I'd suggest to read at least *Section 3.3, "How is data queried"* as it will greatly help you choosing a query approach.

7.1. Using JP-QL

For Hibernate OGM we developed a brand new JP-QL parser which is already able to convert simple queries using Hibernate Search under the precondition that:

- no join, aggregation, or other relational operations are implied
- the target entities and properties are indexed by Hibernate Search

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`

We do realize that this leaves many cases uncovered. So while it might be interesting to try it out, for real usage we suggest for now to use either Hibernate Search full-text queries or the native query technology of the NoSQL storage you are using.

To provide an example of what kind of queries would work:

Example 7.1. Example of trivial Hibernate Query remapped on Hibernate Search

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

```
@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;
}
```

7.2. Using Hibernate Search

We actually did use Hibernate Search already in the previous example; specifically the annotations `@Indexed` and `@Field` are Hibernate Search specific. In this example the query was defined using a JP-QL string and then defining parameters; that's useful if all you have a is a JP-QL Query, but it is limiting.

Hibernate Search remaps the properties annotated with `@Field` in Lucene Documents, and manages the Lucene indexes so that you can then perform Lucene Queries.

To be extremely short, Apache Lucene is a full-text indexing and query engine with excellent query performance. Featurewise, *full-text* means you can do much more than a simple equality match as we did in the previous example.

Let's show another example, now creating a Lucene Query instead:

Example 7.2. Using Hibernate Search for fulltext matching

```
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);
//This is a requirement when using Hibernate OGM instead of ORM:
ftQuery.initializeObjectsWith(ObjectLookupMethod.SKIP,
      DatabaseRetrievalMethod.FIND_BY_ID);

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

Assuming our database contains an `Hypothesis` instance having description "tomorrow we release", the query above will not find the entity because we disabled text analysis in the previous mapping.

If we enable text analysis (which is the default):

Example 7.3. Entity enabling text analysis

```
@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;
}
```

Now the entity would match a query on "tomorrow" as we're unlocking text similarity queries!

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.

For a full explanation of all its capabilities and configuration options, see the *Hibernate Search reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/].

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).

7.3. Using the Criteria API

This is not implemented yet.