

HIBERNATE - Relational Persistence for Idiomatic Java

1

Hibernate Reference Documentation

3.5.6-Final

par Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, et Steve Ebersole

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), Vincent Ricard, Sebastien Cesbron, Michael Courcy, Vincent Giguère, Baptiste Mathus, Emmanuel Bernard, et Anthony Patricio

Préface	xi
1. Tutoriel	1
1.1. Section 1 - Première application Hibernate	1
1.1.1. Configuration	1
1.1.2. La première classe	3
1.1.3. Le fichier de mappage	4
1.1.4. Configuration d'Hibernate	7
1.1.5. Construction avec Maven	10
1.1.6. Démarrage et aides	10
1.1.7. Charger et stocker des objets	11
1.2. Section 2 - Mapper des associations	14
1.2.1. Mapper la classe Person	14
1.2.2. Une association unidirectionnelle basée sur Set	15
1.2.3. Travailler avec l'association	17
1.2.4. Collection de valeurs	19
1.2.5. Associations bidirectionnelles	20
1.2.6. Travailler avec des liens bidirectionnels	21
1.3. Section 3 - L'application web EventManager	22
1.3.1. Écrire la servlet de base	22
1.3.2. Traiter et interpréter	23
1.3.3. Déployer et tester	26
1.4. Résumé	27
2. Architecture	29
2.1. Généralités	29
2.2. Etats des instances	32
2.3. Intégration JMX	32
2.4. Support JCA	33
2.5. Sessions contextuelles	33
3. Configuration	35
3.1. Configuration par programmation	35
3.2. Obtenir une SessionFactory	36
3.3. Connexions JDBC	36
3.4. Propriétés de configuration optionnelles	38
3.4.1. Dialectes SQL	45
3.4.2. Chargement par jointure externe	46
3.4.3. Flux binaires	47
3.4.4. Cache de second niveau et cache de requêtes	47
3.4.5. Substitution dans le langage de requêtes	47
3.4.6. Statistiques Hibernate	47
3.5. Journalisation	47
3.6. Sélectionner une NamingStrategy (stratégie de nommage)	48
3.7. Fichier de configuration XML	49
3.8. Intégration à un serveur d'applications J2EE	50
3.8.1. Configuration de la stratégie transactionnelle	51

3.8.2. SessionFactory associée au JNDI	52
3.8.3. Gestion du contexte de la session courante à JTA	52
3.8.4. Déploiement JMX	53
4. Classes persistantes	55
4.1. Un exemple simple de POJO	55
4.1.1. Implémenter un constructeur sans argument	56
4.1.2. Fournir une propriété d'identifiant (optionnel)	57
4.1.3. Favoriser les classes non finales (optionnel)	57
4.1.4. Déclarer les accesseurs et mutateurs des attributs persistants (optionnel)..	57
4.2. Implémenter l'héritage	58
4.3. Implémenter equals() et hashCode()	58
4.4. Modèles dynamiques	59
4.5. Tuplizers	61
4.6. EntityNameResolvers	63
5. Mappage O/R de base	67
5.1. Déclaration de mappage	67
5.1.1. Doctype	68
5.1.2. Hibernate-mappage	69
5.1.3. Classe	70
5.1.4. id	74
5.1.5. La méthode getter de l'identifiant	78
5.1.6. Optimisation du générateur d'identifiants	80
5.1.7. composite-id	80
5.1.8. Discriminator	82
5.1.9. Version (optionnel)	83
5.1.10. Timestamp (optionnel)	84
5.1.11. Property	85
5.1.12. Plusieurs-à-un	87
5.1.13. Un-à-un	89
5.1.14. Natural-id	91
5.1.15. Component, dynamic-component	92
5.1.16. Propriétés	93
5.1.17. Subclass	94
5.1.18. Joined-subclass	95
5.1.19. Union-subclass	96
5.1.20. Join	97
5.1.21. Key	99
5.1.22. Éléments column et formula	99
5.1.23. Import	100
5.1.24. Any	101
5.2. Types Hibernate	102
5.2.1. Entités et valeurs	102
5.2.2. Types valeurs de base	103
5.2.3. Types de valeur personnalisés	104

5.3. Mapper une classe plus d'une fois	106
5.4. SQL quoted identifiers	106
5.5. Métadonnées alternatives	107
5.5.1. Utilisation de XDoclet	107
5.5.2. Utilisation des annotations JDK 5.0	109
5.6. Propriétés générées	110
5.7. Column read and write expressions	110
5.8. Objets auxiliaires de la base de données	111
6. Mapper une collection	113
6.1. Collections persistantes	113
6.2. Mapper une collection	114
6.2.1. Les clés étrangères d'une collection	116
6.2.2. Les éléments d'une collection	116
6.2.3. Collections indexées	116
6.2.4. Collections de valeurs et associations plusieurs-à-plusieurs	118
6.2.5. Associations un-à-plusieurs	120
6.3. Mappages de collection avancés	121
6.3.1. Collections triées	121
6.3.2. Associations bidirectionnelles	122
6.3.3. Associations bidirectionnelles avec des collections indexées	124
6.3.4. Associations ternaires	125
6.3.5. Using an <idbag>	126
6.4. Exemples de collections	126
7. Mapper les associations	131
7.1. Introduction	131
7.2. Associations unidirectionnelles	131
7.2.1. plusieurs-à-un	131
7.2.2. Un-à-un	132
7.2.3. un-à-plusieurs	133
7.3. Associations unidirectionnelles avec tables de jointure	133
7.3.1. un-à-plusieurs	133
7.3.2. plusieurs-à-un	134
7.3.3. Un-à-un	135
7.3.4. Plusieurs-à-plusieurs	135
7.4. Associations bidirectionnelles	136
7.4.1. un-à-plusieurs / plusieurs-à-un	136
7.4.2. Un-à-un	137
7.5. Associations bidirectionnelles avec tables de jointure	138
7.5.1. un-à-plusieurs / plusieurs-à-un	138
7.5.2. un-à-un	139
7.5.3. Plusieurs-à-plusieurs	140
7.6. Des mappages d'associations plus complexes	141
8. Mappage de composants	143
8.1. Objets dépendants	143

8.2. Collection d'objets dépendants	145
8.3. Les composants en tant qu'indices de Map	147
8.4. Les composants en tant qu'identifiants composites	147
8.5. Les composants dynamiques	149
9. Mapping d'héritage de classe	151
9.1. Les trois stratégies	151
9.1.1. Une table par hiérarchie de classe	151
9.1.2. Une table par classe fille	152
9.1.3. Une table par classe fille, en utilisant un discriminant	153
9.1.4. Mélange d'une table par hiérarchie de classe avec une table par classe fille	153
9.1.5. Une table par classe concrète	154
9.1.6. Une table par classe concrète, en utilisant le polymorphisme implicite	155
9.1.7. Mélange du polymorphisme implicite avec d'autres mappages d'héritage... ..	156
9.2. Limitations	157
10. Travailler avec des objets	159
10.1. États des objets Hibernate	159
10.2. Rendre des objets persistants	159
10.3. Chargement d'un objet	161
10.4. Requêtage	162
10.4.1. Exécution de requêtes	162
10.4.2. Filtrer des collections	166
10.4.3. Requêtes par critères	167
10.4.4. Requêtes en SQL natif	167
10.5. Modifier des objets persistants	168
10.6. Modifier des objets détachés	169
10.7. Détection automatique d'un état	170
10.8. Suppression d'objets persistants	171
10.9. Réplication d'objets entre deux entrepôts de données	171
10.10. Flush de la session	172
10.11. Persistance transitive	173
10.12. Utilisation des méta-données	175
11. Read-only entities	177
11.1. Making persistent entities read-only	177
11.1.1. Entities of immutable classes	178
11.1.2. Loading persistent entities as read-only	178
11.1.3. Loading read-only entities from an HQL query/criteria	179
11.1.4. Making a persistent entity read-only	180
11.2. Read-only affect on property type	181
11.2.1. Simple properties	182
11.2.2. Unidirectional associations	183
11.2.3. Bidirectional associations	185
12. Transactions et Accès concurrents	187
12.1. Portées des sessions et des transactions	187

12.1.1. Unité de travail	188
12.1.2. Longue conversation	189
12.1.3. L'identité des objets	190
12.1.4. Problèmes communs	191
12.2. Démarcation des transactions de base de données	192
12.2.1. Environnement non gérés	193
12.2.2. Utilisation de JTA	194
12.2.3. Gestion des exceptions	196
12.2.4. Timeout de transaction	197
12.3. Contrôle de concurrence optimiste	197
12.3.1. Vérification du versionnage au niveau applicatif	198
12.3.2. Les sessions longues et le versionnage automatique.	198
12.3.3. Les objets détachés et le versionnage automatique	200
12.3.4. Personnaliser le versionnage automatique	200
12.4. Verrouillage pessimiste	201
12.5. Modes de libération de connexion	202
13. Intercepteurs et événements	205
13.1. Intercepteurs	205
13.2. Système d'événements	207
13.3. Sécurité déclarative de Hibernate	208
14. Traitement par lot	211
14.1. Insertions en lot	211
14.2. Mise à jour des lots	212
14.3. L'interface StatelessSession	212
14.4. Opérations de style DML	213
15. HQL : langage d'interrogation d'Hibernate	217
15.1. Sensibilité à la casse	217
15.2. La clause from	217
15.3. Associations et jointures	218
15.4. Formes de syntaxes pour les jointures	220
15.5. Faire référence à la propriété identifiant	220
15.6. La clause select	221
15.7. Fonctions d'agrégation	222
15.8. Requêtes polymorphiques	223
15.9. La clause where	223
15.10. Expressions	225
15.11. La clause order by	229
15.12. La clause group by	230
15.13. Sous-requêtes	231
15.14. Exemples HQL	231
15.15. Nombreuses mises à jour et suppressions	234
15.16. Trucs & Astuces	234
15.17. Composants	235
15.18. Syntaxe des constructeurs de valeur de ligne	236

16. Requêtes par critères	239
16.1. Créer une instance de Criteria	239
16.2. Restriction du résultat	239
16.3. Trier les résultats	240
16.4. Associations	241
16.5. Peuplement d'associations de manière dynamique	242
16.6. Requêtes par l'exemple	242
16.7. Projections, agrégation et regroupement	243
16.8. Requêtes et sous-requêtes détachées	245
16.9. Requêtes par identifiant naturel	246
17. SQL natif	247
17.1. Utiliser une requête SQLQuery	247
17.1.1. Requêtes scalaires	247
17.1.2. Requêtes d'entités	248
17.1.3. Gérer les associations et collections	249
17.1.4. Retour d'entités multiples	249
17.1.5. Retour d'entités non gérées	251
17.1.6. Gérer l'héritage	252
17.1.7. Paramètres	252
17.2. Requêtes SQL nommées	252
17.2.1. Utilisation de return-property pour spécifier explicitement les noms des colonnes/alias	254
17.2.2. Utilisation de procédures stockées pour les requêtes	255
17.3. SQL personnalisé pour créer, mettre à jour et effacer	256
17.4. SQL personnalisé pour le chargement	258
18. Filtrer les données	261
18.1. Filtres Hibernate	261
19. Mappage XML	265
19.1. Travailler avec des données XML	265
19.1.1. Spécifier le mappage XML et le mappage d'une classe ensemble	265
19.1.2. Spécifier seulement un mappage XML	266
19.2. Métadonnées du mappage XML	266
19.3. Manipuler des données XML	268
20. Améliorer les performances	271
20.1. Stratégies de chargement	271
20.1.1. Travailler avec des associations chargées en différé	272
20.1.2. Personnalisation des stratégies de chargement	273
20.1.3. Proxies pour des associations vers un seul objet	274
20.1.4. Initialisation des collections et des proxies	276
20.1.5. Utiliser le chargement par lot	277
20.1.6. Utilisation du chargement par sous select	278
20.1.7. Fetch profiles	278
20.1.8. Utiliser le chargement en différé des propriétés	280
20.2. Le cache de second niveau	281

20.2.1. Mappages de Cache	282
20.2.2. Stratégie : lecture seule	282
20.2.3. Stratégie : lecture/écriture	283
20.2.4. Stratégie : lecture/écriture non stricte	283
20.2.5. Stratégie : transactionnelle	283
20.2.6. Support de stratégie de concurrence du fournisseur-cache	284
20.3. Gérer les caches	284
20.4. Le cache de requêtes	286
20.4.1. Enabling query caching	286
20.4.2. Query cache regions	287
20.5. Comprendre les performances des collections	287
20.5.1. Taxinomie	287
20.5.2. Les lists, les maps, les idbags et les ensembles sont les collections les plus efficaces pour la mise à jour	288
20.5.3. Les sacs et les listes sont les plus efficaces pour les collections inverses.	289
20.5.4. Suppression en un coup	289
20.6. Moniteur de performance	290
20.6.1. Suivi d'une SessionFactory	290
20.6.2. Métriques	291
21. Guide de la boîte à outils	293
21.1. Génération automatique du schéma	293
21.1.1. Personnaliser le schéma	294
21.1.2. Exécuter l'outil	297
21.1.3. Propriétés	297
21.1.4. Utiliser Ant	298
21.1.5. Mises à jour incrémentales du schéma	298
21.1.6. Utiliser Ant pour des mises à jour de schéma par incrément	299
21.1.7. Validation du schéma	299
21.1.8. Utiliser Ant pour la validation du Schéma	300
22. Exemple : père/fils	301
22.1. Une note à propos des collections	301
22.2. Un-à-plusieurs bidirectionnel	301
22.3. Cycle de vie en cascade	303
22.4. Cascades et unsaved-value (valeurs non sauvegardées)	305
22.5. Conclusion	305
23. Exemple : application Weblog	307
23.1. Classes persistantes	307
23.2. Mappages Hibernate	308
23.3. Code Hibernate	310
24. Exemple : quelques mappages	315
24.1. Employeur/Employé (Employer/Employee)	315
24.2. Auteur/Travail	317
24.3. Client/Commande/Produit	319
24.4. Divers exemples de mappages	321

24.4.1. "Typed" association un-à-un	321
24.4.2. Exemple de clef composée	321
24.4.3. Plusieurs-à-plusieurs avec un attribut de clef composée partagée	324
24.4.4. Contenu basé sur une discrimination	324
24.4.5. Associations sur des clés alternées	325
25. Meilleures pratiques	327
26. Considérations de portabilité des bases de données	331
26.1. Aspects fondamentaux de la portabilité	331
26.2. Dialecte	331
26.3. Résolution de dialecte	331
26.4. Générer les identifiants	332
26.5. Fonctions de base de données	333
26.6. Type mappings	333
References	335

Préface

Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. Hibernate is an Object/Relational Mapping tool for Java environments. The term Object/Relational Mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

Hibernate s'occupe du mappage des classes Java vers les tables de bases de données (et des types de données Java vers les types de données SQL), mais fournit également des facilités de recherche et de retrait de données. Hibernate peut réduire énormément le temps de développement, normalement passé à traiter des données manuellement dans SQL ou JDBC.

Le but d'Hibernate est de libérer le développeur de 95 pour cent des tâches de programmation liées à la persistance de données communes. Hibernate n'est peut-être pas la meilleure solution pour les applications centrées-données qui utilisent uniquement les procédures-stored pour implémenter la logique métier dans la base de données. Cela est surtout utile avec les modèles de domaines orientés-objet et la logique métier dans l'étape intermédiaire basée Java. Malgré tout, Hibernate peut vous aider à supprimer ou à encapsuler le code SQL propre à un distributeur et vous aidera à régler la tâche commune qui consiste à transposer un ensemble de résultats à partir d'un tableau de représentation vers un graphe d'objets.

Si vous n'êtes pas familiarisé avec Hibernate et le mappage Objet/Relationnel ou même Java, veuillez suivre les étapes suivantes :

1. Read [Chapitre 1, Tutoriel](#) for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.
2. Read [Chapitre 2, Architecture](#) to understand the environments where Hibernate can be used.
3. Veuillez consulter le répertoire `eg/` dans la distribution Hibernate, qui contient une application autonome simple. Copier votre pilote JDBC dans le répertoire `lib/` et éditez `etc/hibernate.properties`, en spécifiant les valeurs qu'il faut dans votre base de données. A partir d'une invite de commande du répertoire de distribution, veuillez saisir `ant eg` (en utilisant Ant), et sous Windows, tapez `build eg`.
4. Use this reference documentation as your primary source of information. Consider reading [\[JPwH\]](#) if you need more help with application design, or if you prefer a step-by-step tutorial. Also visit <http://caveatemptor.hibernate.org> and download the example application from [\[JPwH\]](#).
5. Les questions FAQ sont traitées sur le site Hibernate.
6. Links to third party demos, examples, and tutorials are maintained on the Hibernate website.
7. La section Community Area (Zône communautaire) du site Hibernate constitue une ressource intéressante pour les modèles conceptuels et autres solutions diverses d'intégration (Tomcat, JBoss AS, Struts, EJB, etc.).

Si vous avez des questions, participez au forum utilisateur sur le site Hibernate. Nous proposons également des systèmes de traçage JIRA pour les rapports de bogues et les demandes sur les fonctionnalités. Si vous êtes intéressé à participer au développement d'Hibernate, veuillez rejoindre la liste de distribution électronique des développeurs.

Le support pour le développement commercial, le support production, et la formation Hibernate sont disponibles à travers JBoss Inc. (voir <http://www.hibernate.org/SupportTraining/>). Hibernate est un projet professionnel en source ouverte et un composant critique de la suite de produits JBoss Enterprise Middleware System (JEMS).

Tutoriel

A l'intention des nouveaux utilisateurs, ce chapitre fournit une introduction étape par étape à Hibernate, en commençant par une application simple, avec une base de données en-mémoire. Le tutoriel est basé sur un tutoriel antérieur qui avait été développé par Michael Gloegl. Tout le code est contenu dans `tutorials/web` qui se trouve dans le répertoire source du projet.



Important

Ce tutoriel assume que l'utilisateur est déjà familier avec Java et SQL à la fois. Si vous ne possédez qu'une connaissance de Java et d'SQL limitée, il est conseillé de commencer par vous familiariser avec ces technologies avant d'aborder Hibernate.



Note

La distribution contient un autre exemple d'application qui se trouve dans le répertoire source du projet `tutorial/eg`.

1.1. Section 1 - Première application Hibernate

Supposons que nous ayons besoin d'une petite application de base de données qui puisse stocker des événements que nous voulons suivre, et des informations à propos des hôtes de ces événements.



Note

Malgré que vous puissiez utiliser toute base de données qui vous convienne, on choisira [HSQLDB](http://hsqldb.org/) [http://hsqldb.org/] (une base de données Java, en-mémoire) pour éviter de décrire l'installation et la configuration de n'importe quel serveur de base de données particulière.

1.1.1. Configuration

La première chose que nous devons faire est de configurer l'environnement de développement. Nous utiliserons la "standard layout" préconisée par de nombreux outils de génération tels que [Maven](http://maven.org) [http://maven.org]. Maven, en particulier, a une bonne ressource décrivant cette [layout](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html) [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]. Comme ce tutoriel va devenir une application web, nous allons créer et utiliser les répertoires `src/main/java`, `src/main/ressources` et `src/main/webapp`.

Nous utiliserons Maven dans ce tutoriel. Nous profiterons de ses capacités de gestion de dépendances transitives, ainsi que de la capacité des nombreux IDE à installer automatiquement un projet sur la base du descripteur Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion>
>4.0.0</modelVersion>

    <groupId>
>org.hibernate.tutorials</groupId>
    <artifactId>
>hibernate-tutorial</artifactId>
    <version>
>1.0.0-SNAPSHOT</version>
    <name>
>First Hibernate Tutorial</name>

    <build>
        <!-- we dont want the version to be part of the generated war file name -->
        <finalName>
>${artifactId}</finalName>
    </build>

    <dependencies>
        <dependency>
            <groupId>
>org.hibernate</groupId>
            <artifactId>
>hibernate-core</artifactId>
        </dependency>

        <!-- Because this is a web app, we also have a dependency on the servlet api. -->
        <dependency>
            <groupId>
>javax.servlet</groupId>
            <artifactId>
>servlet-api</artifactId>
        </dependency>

        <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
        <dependency>
            <groupId>
>org.slf4j</groupId>
            <artifactId>
>slf4j-simple</artifactId>
        </dependency>

        <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
        <dependency>
            <groupId>
>javassist</groupId>
            <artifactId>
>javassist</artifactId>
```

```
        </dependency>
    </dependencies>

</project>
>
```



Astuce

It is not a requirement to use Maven. If you wish to use something else to build this tutorial (such as Ant), the layout will remain the same. The only change is that you will need to manually account for all the needed dependencies. If you use something like [Ivy](http://ant.apache.org/ivy/) [http://ant.apache.org/ivy/] providing transitive dependency management you would still use the dependencies mentioned below. Otherwise, you'd need to grab *all* dependencies, both explicit and transitive, and add them to the project's classpath. If working from the Hibernate distribution bundle, this would mean `hibernate3.jar`, all artifacts in the `lib/required` directory and all files from either the `lib/bytecode/cglib` or `lib/bytecode/javassist` directory; additionally you will need both the `servlet-api` jar and one of the `slf4j` logging backends.

Sauvegardez ce fichier sous la forme `pom.xml` dans le répertoire root du projet.

1.1.2. La première classe

Ensuite, nous créons une classe qui représente l'évènement que nous voulons stocker dans notre base de données. Il s'agit d'une simple classe `JavaBean` avec quelques propriétés :

```
package org.hibernate.tutorial.domain;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

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

    public Date getDate() {
        return date;
    }
}
```

```
}

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Vous constaterez que cette classe utilise les conventions de nommage standard JavaBean pour les méthodes getter/setter des propriétés, ainsi qu'une visibilité privée pour les champs. Ceci est la conception recommandée - mais pas obligatoire. Hibernate peut aussi accéder aux champs directement, le bénéfice des méthodes d'accès est la robustesse pour la refonte de code.

La propriété `id` contient la valeur d'un identifiant unique pour un événement particulier. Toutes les classes d'entités persistantes (il y a également des classes dépendantes de moindre importance) auront besoin d'une telle propriété identifiante si nous voulons utiliser l'ensemble complet des fonctionnalités de Hibernate. En fait, la plupart des applications (surtout les applications web) ont besoin de distinguer des objets par des identifiants, par conséquent considérez cela comme une fonctionnalité et non comme une limitation. Cependant, nous ne manipulons généralement pas l'identité d'un objet, dorénavant la méthode setter devrait être privée. Seul Hibernate assignera les identifiants lorsqu'un objet est sauvegardé. Remarquez que Hibernate peut accéder aux méthodes publiques, privées et protégées, ainsi qu'aux champs (publics, privés, protégés) directement. À vous de choisir, et vous pouvez également l'ajuster à la conception de votre application.

Le constructeur sans argument est requis pour toutes les classes persistantes; Hibernate doit créer des objets pour vous en utilisant la réflexion Java. Le constructeur peut être privé, cependant, la visibilité du paquet est requise pour la génération de proxies à l'exécution et une récupération efficace des données sans instrumentation du bytecode.

Sauvegardez ce fichier dans le répertoire `src/main/java/org/hibernate/tutorial/domain`.

1.1.3. Le fichier de mappage

Hibernate a besoin de savoir comment charger et stocker des objets d'une classe persistante. C'est là qu'intervient le fichier de mappage Hibernate. Le fichier de mappage indique à Hibernate à quelle table accéder dans la base de données, et les colonnes de cette table à utiliser.

La structure basique de ce fichier de mappage ressemble à ce qui suit :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```



```

    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.domain">
[... ]
</hibernate-mapping
>

```

Notez que la DTD Hibernate est très sophistiquée. Vous pouvez l'utiliser pour l'auto-finalisation des éléments et des attributs de mappage XML dans votre éditeur ou votre IDE. Ouvrez également le fichier DTD dans votre éditeur de texte - c'est le moyen le plus facile d'obtenir une vue d'ensemble de tous les éléments et attributs, et de voir les valeurs par défaut, ainsi que quelques commentaires. Notez qu'Hibernate ne chargera pas le fichier DTD à partir du web, mais regardera d'abord dans le chemin de classe de l'application. Le fichier DTD est inclus dans `hibernate-core.jar` ainsi que dans le répertoire `src` de la distribution Hibernate).



Important

Nous omettrons la déclaration de la DTD dans les exemples futurs pour raccourcir le code. Évidemment il n'est pas optionnel.

Entre les deux balises `hibernate-mapping`, incluez un élément `class`. Toutes les classes d'entités persistantes (encore une fois, il pourrait y avoir des classes dépendantes plus tard, qui ne sont pas des entités mère) ont besoin d'un mappage vers une table de la base de données SQL :

```

<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">

    </class>

</hibernate-mapping
>

```

Plus loin, nous indiquons à Hibernate comment persister et charger un objet de la classe `Event` dans la table `EVENTS`, chaque instance étant représentée par une ligne dans cette table. Maintenant nous continuons avec le mappage de la propriété de l'identifiant unique vers la clé primaire des tables. De plus, comme nous ne voulons pas nous occuper de la gestion de cet identifiant, nous utilisons une stratégie de génération d'identifiant Hibernate pour la colonne de la clé primaire subrogée :

```

<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>

```

```
        </id>
    </class>

</hibernate-mapping>
>
```

L'élément `ID` est la déclaration de l'identifiant de propriété. L'attribut de mappage `name="id"` déclare le nom de la propriété JavaBean et indique à Hibernate d'utiliser les méthodes `getId()` et `setId()` pour accéder à la propriété. L'attribut de colonne indique à Hibernate quelle colonne de la table `EVENTS` contient la valeur de clé primaire.

L'élément imbriqué `Générateur` spécifie la stratégie de génération d'identifiant (c'est à dire comment les valeurs d'identifiant sont-elles générées?). Dans ce cas nous avons choisi `native`, qui offre un niveau de la portabilité selon le dialecte de base de données configurée. Mise en veille prolongée prend en charge la base de données générée, unique au monde, ainsi que l'application affectée, les identifiants. Génération de valeur d'identifiant est aussi l'un des nombreux points d'extension d'Hibernate et vous pouvez plug-in votre propre stratégie.



Astuce

`native` is no longer consider the best strategy in terms of portability. for further discussion, see [Section 26.4](#), « *Générer les identifiants* »

Enfin, nous incluons des déclarations pour les propriétés persistantes de la classe dans le fichier de mappage. Par défaut, aucune propriété de la classe n'est considérée comme persistante :

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
>
```

Comme avec l'élément `id`, l'attribut `name` de l'élément `property` indique à Hibernate quelles méthodes getters/setters utiliser. Par conséquent dans ce cas, Hibernate cherchera `getDate()`/`setDate()`, de même que `getTitle()`/`setTitle()`.



Note

Pourquoi le mappage de la propriété `date` inclut-il l'attribut `column`, mais non le `title` ? Sans l'attribut `column`, Hibernate utilise par défaut le nom de la propriété comme nom de colonne. Cela fonctionne bien pour `title`. Cependant, `date` est un mot clé réservé dans la plupart des bases de données, donc nous utilisons un nom différent pour le mappage.

Il est intéressant de noter que le mappage de `title` manque également d'un attribut `type`. Les types que nous déclarons et utilisons dans les fichiers de mappage ne sont pas, comme vous pourriez vous y attendre, des types de données Java. Ce ne sont pas, non plus, des types de base de données SQL. Ces types sont donc appelés *types de mappage Hibernate*, des convertisseurs qui peuvent traduire des types Java en types SQL et vice versa. De plus, Hibernate tentera de déterminer la bonne conversion et le type de mappage lui-même si l'attribut `type` n'est pas présent dans le mappage. Dans certains cas, cette détection automatique (utilisant la réflexion sur la classe Java) pourrait ne pas donner la valeur attendue ou dont vous avez besoin. C'est le cas avec la propriété `date`. Hibernate ne peut pas savoir si la propriété "mappera" une colonne SQL de type `date`, `timestamp` ou `time`. Nous déclarons que nous voulons conserver des informations avec une date complète et l'heure en mappant la propriété avec un convertisseur `timestamp`.



Astuce

Hibernate rend cette détermination de type de mappage en utilisant la réflexion au moment du traitement des fichiers de mappage. Cela prend du temps et consomme des ressources, donc, si la performance de démarrage est importante, vous devriez considérer définir explicitement quel type utiliser.

Sauvegardez ce fichier de mappage ainsi `src/main/resources/org/hibernate/tutorial/domain/Event.hbm.xml`.

1.1.4. Configuration d'Hibernate

A ce niveau là, vous devriez avoir la classe persistante et son fichier de mappage en place. Il est temps maintenant de configurer Hibernate. Tout d'abord, il nous faut configurer HSQLDB pour qu'il puisse exécuter en "server mode"



Note

We do this do that the data remains between runs.

Vous utiliserez le lugin `exec Maven` pour lancer le serveur HSQLDB en exécutant :

```
mvn exec:java -Dexec.mainClass="org.hsqldb.Server" -Dexec.args="-database.0"
```

file:target/data/tutorial". Vous observez qu'elle démarre et ouvre un socket TCP/IP, c'est là que notre application se connectera plus tard. Si vous souhaitez démarrer à partir d'une nouvelle base de données pour ce tutoriel (choisissez `CTRL + C` dans la fenêtre), effacez tous les fichiers dans le répertoire `target/data` et redémarrez HSQL DB.

Hibernate se connectera à la base de données pour le compte de votre application, donc il devra savoir comment obtenir des connexions. Pour ce tutoriel, nous devrons utiliser un pool de connexions autonomes (et non pas `javax.sql.DataSource`). Hibernate bénéficie du support de deux pools de connexions JDBC open source de tierce partie : [c3p0](https://sourceforge.net/projects/c3p0) [https://sourceforge.net/projects/c3p0] and [proxool](http://proxool.sourceforge.net/) [http://proxool.sourceforge.net/]. Cependant, nous utiliserons le pool de connexions intégré Hibernate pour ce tutoriel.



Attention

The built-in Hibernate connection pool is in no way intended for production use. It lacks several features found on any decent connection pool.

Pour la configuration de Hibernate, nous pouvons utiliser un simple fichier `hibernate.properties`, un fichier `hibernate.cfg.xml` légèrement plus sophistiqué, ou même une configuration complète par programmation. La plupart des utilisateurs préfèrent le fichier de configuration XML :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class"
>org.hsqldb.jdbcDriver</property>
        <property name="connection.url"
>jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username"
>sa</property>
        <property name="connection.password"
></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size"
>1</property>

        <!-- SQL dialect -->
        <property name="dialect"
>org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class"
```

```

>thread</property>

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class"
>org.hibernate.cache.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql"
>true</property>

    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto"
>update</property>

    <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml" />

</session-factory>

</hibernate-configuration
>

```



Note

Vous pourrez remarquer que cette configuration XML utilise une DTD différente.

Nous configurons une `SessionFactory` de Hibernate - une fabrique globale responsable d'une base de données particulière. Si vous avez plusieurs base de données, utilisez plusieurs configurations `<session-factory>`, généralement dans des fichiers de configuration différents (pour un démarrage plus facile).

Les quatre premiers éléments `property` contiennent la configuration nécessaire pour la connexion JDBC. L'élément `property` du dialecte spécifie quelle variante du SQL Hibernate va générer.



Astuce

In most cases, Hibernate is able to properly determine which dialect to use. See [Section 26.3, « Résolution de dialecte »](#) for more information.

La gestion automatique des sessions d'Hibernate pour les contextes de persistance est bien pratique, comme vous pourrez le constater. L'option `hbm2ddl.auto` active la génération automatique des schémas de base de données - directement dans la base de données. Cela peut également être désactivé (en supprimant l'option de configuration) ou redirigé vers un fichier avec l'aide de la tâche Ant `SchemaExport`. Finalement, nous ajoutons le(s) fichier(s) de mappage pour les classes persistantes.

Sauvegarder ce fichier en tant que `hibernate.cfg.xml` dans le répertoire `src/main/resources`.

1.1.5. Construction avec Maven

Nous allons maintenant construire le tutoriel avec Maven. Vous aurez besoin d'installer Maven pour cela. Il est disponible dans la page [Maven download page](http://maven.apache.org/download.html) [http://maven.apache.org/download.html]. Maven pourra lire le fichier `/pom.xml` que nous avons créé plus tôt et saura comment effectuer quelques tâches du projet de base. Tout d'abord, exécutons `compile` pour s'assurer que nous pouvons tout compiler jusqu'à maintenant :

```
[hibernateTutorial]$ mvn compile
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building First Hibernate Tutorial
[INFO]    task-segment: [compile]
[INFO] -----
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /home/steve/projects/sandbox/hibernateTutorial/target/classes
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jun 09 12:25:25 CDT 2009
[INFO] Final Memory: 5M/547M
[INFO] -----
```

1.1.6. Démarrage et aides

Il est temps de charger et de stocker quelques objets `Event`, mais d'abord nous devons compléter la configuration avec du code d'infrastructure. Nous devons démarrer Hibernate. Ce démarrage inclut la construction d'un objet `SessionFactory` global et le stocker dans un lieu facile d'accès dans le code de l'application. Une `SessionFactory` peut ouvrir de nouvelles `Sessions`. Une `Session` représente une unité de travail simplement "threadée". La `org.hibernate.SessionFactory` est un objet global "thread-safe", instancié une seule fois.

Nous créerons une classe d'aide `HibernateUtil` qui s'occupe du démarrage et rend la gestion des `org.hibernate.SessionFactory` plus facile.

```
package org.hibernate.tutorial.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml

```

```

        return new Configuration().configure().buildSessionFactory();
    }
    catch (Throwable ex) {
        // Make sure you log the exception, as it might be swallowed
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}

```

Sauvegardez ce code en tant que `src/main/java/org/hibernate/tutorial/util/HibernateUtil.java`

Cette classe ne produit pas seulement la `org.hibernate.SessionFactory` globale dans un initialiseur statique. Elle masque le fait qu'elle exploite un singleton statique. Nous aurions pu aussi bien verrouiller la référence `org.hibernate.SessionFactory` à partir de JNDI dans un serveur d'application ou dans n'importe quelle location en fait. Elle pourrait aussi obtenir la `SessionFactory` depuis JNDI dans un serveur d'applications.

Si vous nommez `org.hibernate.SessionFactory` dans votre fichier de configuration, Hibernate tentera la récupération depuis JNDI. Pour éviter ce code, vous pouvez aussi utiliser un déploiement JMX et laisser le conteneur (compatible JMX) instancier et lier un `HibernateService` à JNDI. Ces options avancées sont expliquées plus loin.

Nous avons finalement besoin de configurer le système de journalisation - Hibernate utilise commons-logging et vous laisse le choix entre log4j et le système de logs du JDK 1.4. La plupart des développeurs préfèrent log4j : copiez `log4j.properties` de la distribution de Hibernate (il est dans le répertoire `etc/`) dans votre répertoire `src`, puis faites de même avec `hibernate.cfg.xml`. Regardez la configuration d'exemple et changez les paramètres si vous voulez une sortie plus verbeuse. Par défaut, seul le message de démarrage de Hibernate est affiché sur la sortie standard.

L'infrastructure de ce tutorial est complète - et nous sommes prêts à effectuer un travail réel avec Hibernate.

1.1.7. Charger et stocker des objets

We are now ready to start doing some real work with Hibernate. Let's start by writing an `EventManager` class with a `main()` method:

```

package org.hibernate.tutorial;

import org.hibernate.Session;

```

```
import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);

        session.getTransaction().commit();
    }
}
```

Nous créons un nouvel objet `Event` dans `createAndStoreEvent()`, et nous le remettons à `Hibernate`, qui s'occupe maintenant du SQL et exécute les `INSERT` s dans la base de données.

A `org.hibernate.Session` is designed to represent a single unit of work (a single atomic piece of work to be performed). For now we will keep things simple and assume a one-to-one granularity between a `Hibernate org.hibernate.Session` and a database transaction. To shield our code from the actual underlying transaction system we use the `Hibernate org.hibernate.Transaction` API. In this particular case we are using JDBC-based transactional semantics, but it could also run with JTA.

Quelle est la fonction de `sessionFactory.getCurrentSession()` ? Premièrement, vous pouvez l'invoquer autant de fois que vous le voulez et n'importe où, du moment que vous avez votre `SessionFactory` (facile grâce à `HibernateUtil`). La méthode `getCurrentSession()` renvoie toujours l'unité de travail courante. Souvenez vous que nous avons basculé notre option de configuration au mécanisme basé sur le "thread" dans `hibernate.cfg.xml`. Par conséquent, l'unité de travail courante est liée au thread Java courant qui exécute notre application.



Important

Hibernate offre trois méthodes le suivi de session courant. La méthode basée "thread" qui n'est pas conçue pour une utilisation de la production ; seulement utile

pour les prototypes et des tutoriels comme celui-ci. Le suivi de session courant est abordé plus en détail par la suite.

Une `org.hibernate.Session` commence lorsque le thread courant commence à appeler `getCurrentSession()`. Ensuite, elle est attachée par Hibernate au thread courant. Lorsque la transaction s'achève, par `commit` ou par `rollback`, Hibernate détache automatiquement la `Session` du thread et la ferme pour vous. Si vous invoquez `getCurrentSession()` une nouvelle fois, vous obtenez une nouvelle `Session` et pouvez entamer une nouvelle unité de travail.

A propos de la portée de l'unité de travail, la session `org.hibernate.Session` Hibernate devrait-elle être utilisée pour exécuter une ou plusieurs opérations en base de données ? L'exemple ci-dessus utilise une `Session` pour une opération. C'est une pure coïncidence, l'exemple n'est pas assez complexe pour montrer d'autres approches. La portée d'une `Session` Hibernate est flexible mais vous ne devriez jamais concevoir votre application de manière à utiliser une nouvelle `Session` Hibernate pour *chaque* opération en base de données. Donc même si vous le voyez quelquefois dans les exemples suivants, considérez *une session par opération* comme un anti-modèle. Une véritable application (web) est affichée plus loin dans ce tutoriel.

See [Chapitre 12, Transactions et Accès concurrents](#) for more information about transaction handling and demarcation. The previous example also skipped any error handling and rollback.

Pour pouvoir exécuter ceci, nous utiliserons le plugin `exec` Maven pour appeler notre classe avec la configuration de classpath qui convient : `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="store"`



Note

Vous aurez sans doute besoin d'effectuer `mvn compile` pour commencer.

Vous devriez constater qu'Hibernate démarre et selon votre configuration, beaucoup de traces sur la sortie. À la fin, vous trouverez la ligne suivante :

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

C'est l' `INSERT` exécutée par Hibernate.

Maintenant nous aimerions aussi lister les événements stockés, donc nous ajoutons une option à la méthode principale :

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
}
```

```
for (int i = 0; i < events.size(); i++) {
    Event theEvent = (Event) events.get(i);
    System.out.println(
        "Event: " + theEvent.getTitle() + " Time: " + theEvent.getDate()
    );
}
```

Nous ajoutons aussi une nouvelle méthode `listEvents()` :

```
private List listEvents() {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    List result = session.createQuery("from Event").list();
    session.getTransaction().commit();
    return result;
}
```

Here, we are using a Hibernate Query Language (HQL) query to load all existing `Event` objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate `Event` objects with the data. You can create more complex queries with HQL. See [Chapitre 15, HQL : langage d'interrogation d'Hibernate](#) for more information.

Nous pouvons maintenant appeler notre nouvelle fonctionnalité, en utilisant à nouveau le plugin `exec Maven` : `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="list"`

1.2. Section 2 - Mapper des associations

Pour l'instant, nous nous sommes contentés de mapper une classe d'une entité persistante vers une table. Profitons-en pour ajouter quelques associations de classe. D'abord nous ajouterons des gens à notre application, et stockerons une liste d'événements auxquels ils participent.

1.2.1. Mapper la classe `Person`

La première version de la classe `Person` est simple :

```
package org.hibernate.tutorial.domain;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'
```

```
}
```

A sauvegarder dans le fichier nommé `src/main/java/org/hibernate/tutorial/domain/Person.java`

Puis, créez le nouveau fichier de mappage `src/main/resources/org/hibernate/tutorial/domain/Person.hbm.xml`

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
>
```

Finalement, ajoutez le nouveau mappage à la configuration d'Hibernate :

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Nous allons maintenant créer une association entre ces deux entités. Évidemment, des personnes peuvent participer aux événements, et des événements ont des participants. Les questions de conception que nous devons traiter sont : direction, cardinalité et comportement de la collection.

1.2.2. Une association unidirectionnelle basée sur Set

Nous allons ajouter une collection d'événements à la classe `Person`. De cette manière nous pouvons facilement naviguer dans les événements d'une personne particulière, sans exécuter une requête explicite - en appelant `aPerson.getEvents()`. Nous utilisons une collection Java, un `Set`, parce que la collection ne contiendra pas d'éléments dupliqués et l'ordre ne nous importe pas pour ces exemples :

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }
}
```

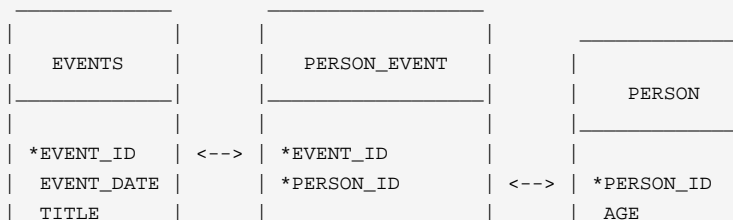
```
public void setEvents(Set events) {  
    this.events = events;  
}  
}
```

D'abord nous mappons cette association, mais pensez à l'autre côté. Clairement, nous pouvons la laisser unidirectionnelle. Ou bien, nous pourrions créer une autre collection sur `Event`, si nous voulons être capable de la parcourir de manière bidirectionnelle. Ce n'est pas nécessaire d'un point de vue fonctionnel. Vous pourrez toujours exécuter une requête explicite pour récupérer les participants d'un événement particulier. Vous êtes libre de choisir la conception, ce qui est certain, c'est que la cardinalité de l'association : "plusieurs" valus des deux côtés, est appelée *plusieurs-à-plusieurs*. Par conséquent nous utilisons un mappage Hibernate plusieurs-à-plusieurs :

```
<class name="Person" table="PERSON">  
    <id name="id" column="PERSON_ID">  
        <generator class="native"/>  
    </id>  
    <property name="age"/>  
    <property name="firstname"/>  
    <property name="lastname"/>  
  
    <set name="events" table="PERSON_EVENT">  
        <key column="PERSON_ID"/>  
        <many-to-many column="EVENT_ID" class="Event"/>  
    </set>  
  
</class>  
>
```

Hibernate supporte toutes sortes de mappage de collection, un `set` étant le plus commun. Pour une association plusieurs-à-plusieurs (ou une relation d'entité $n:m$), une table d'association est requise. Chaque ligne dans cette table représente un lien entre une personne et un événement. Le nom de la table est configuré avec l'attribut `table` de l'élément `set`. Le nom de la colonne identifiant dans l'association, du côté de la personne, est défini avec l'élément `key`, et le nom de la colonne pour l'événement avec l'attribut `column` de `many-to-many`. Vous devez aussi donner à Hibernate la classe des objets de votre collection (c'est-à-dire : la classe de l'autre côté de la collection).

Le schéma de base de données pour ce mappage est donc :



_____	FIRSTNAME
	LASTNAME

1.2.3. Travailler avec l'association

Réunissons quelques personnes et quelques événements dans une nouvelle méthode dans `EventManager` :

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);
    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

Après le chargement d'une `Person` et d'un `Event`, modifiez simplement la collection en utilisant les méthodes normales de la collection. Comme vous pouvez le constater, il n'y a pas d'appel explicite à `update()` ou `save()`, Hibernate détecte automatiquement que la collection a été modifiée et a besoin d'être mise à jour. Ceci est appelé *la vérification sale automatique* (automatic dirty checking), et vous pouvez aussi l'essayer en modifiant le nom ou la propriété date de n'importe lequel de vos objets. Tant qu'ils sont dans un état *persistant*, c'est-à-dire, liés à une `Session` Hibernate particulière (c-à-d qu'ils ont juste été chargés ou sauvegardés dans une unité de travail), Hibernate surveille les changements et exécute le SQL correspondants. Le processus de synchronisation de l'état de la mémoire avec la base de données, généralement seulement à la fin d'une unité de travail, est appelé *flushing*. Dans notre code, l'unité de travail s'achève par un `commit` (ou `rollback`) de la transaction avec la base de données.

Vous pourriez bien sûr charger une personne et un événement dans différentes unités de travail. Ou vous modifiez un objet à l'extérieur d'une `Session`, s'il n'est pas dans un état persistant (s'il était persistant avant, nous appelons cet état *détaché*). Vous pouvez même modifier une collection lorsqu'elle est détachée :

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
        .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
        .setParameter("pid", personId)
        .uniqueResult(); // Eager fetch the collection so we can use it detached
    Event anEvent = (Event) session.load(Event.class, eventId);
```

```
session.getTransaction().commit();

// End of first unit of work

aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

// Begin second unit of work

Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
session2.beginTransaction();
session2.update(aPerson); // Reattachment of aPerson

session2.getTransaction().commit();
}
```

L'appel à `update` rend un objet détaché à nouveau persistant, vous pourriez dire qu'il le lie à une nouvelle unité de travail, ainsi toutes les modifications que vous avez faites pendant qu'il était détaché peuvent être sauvegardées dans la base de données, cela inclut toute modification effectuées sur une collection de cet objet entité.

Cela n'a pas grand intérêt dans notre situation, mais c'est un concept important qu'il vous faut concevoir dans votre application. Pour le moment, complétez cet exercice en ajoutant une nouvelle action à la méthode principale de l'`EventManager` et invoquez-la depuis la ligne de commande. Si vous avez besoin des identifiants d'un client et d'un évènement - la méthode `save()` vous les retourne (vous devrez peut-être modifier certaines méthodes précédentes pour retourner ces identifiants) :

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date());
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " + eventId);
}
```

C'était un exemple d'une association entre deux classes de même importance, deux entités. Comme mentionné plus tôt, il y a d'autres classes et d'autres types dans un modèle typique, généralement "moins importants". Vous en avez déjà vu certains, comme un `int` ou une `String`. Nous appelons ces classes des *types de valeur*, et leurs instances *dépendent* d'une entité particulière. Des instances de ces types n'ont pas leur propre identité, elles ne sont pas non plus partagées entre des entités (deux personnes ne référencent pas le même objet `firstname`, même si elles ont le même prénom). Bien sûr, des types de valeur n'existent pas seulement dans le JDK (en fait, dans une application Hibernate toutes les classes du JDK sont considérées comme des types de valeur), vous pouvez aussi écrire vous-même des classes dépendantes, `Address` ou `MonetaryAmount`, par exemple.

Vous pouvez aussi concevoir une collection de types de valeur. C'est conceptuellement très différent d'une collection de références vers d'autres entités, mais très ressemblant dans Java.

1.2.4. Collection de valeurs

Ajoutons un ensemble d'adresses email à l'entité `Person` qui sera représenté en tant que `java.util.Set` d'instance `java.lang.String` :

```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

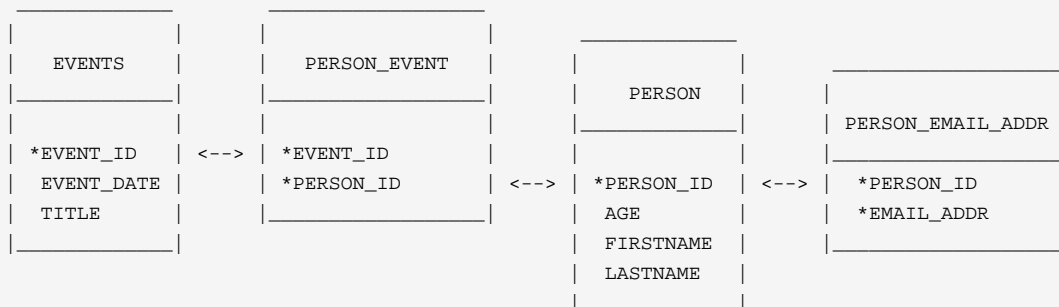
public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

Le mappage de ce `Set` :

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
>
```

La différence comparée au mappage vu plus tôt est la partie `element`, qui indique à Hibernate que la collection ne contient pas de référence vers une autre entité, mais une collection d'éléments de type `String` (le nom en minuscule vous indique que c'est un type/convertisseur du mappage Hibernate). Une fois encore, l'attribut `table` de l'élément `set` détermine le nom de la table pour la collection. L'élément `key` définit le nom de la colonne de la clé étrangère dans la table de la collection. L'attribut `column` dans l'élément `element` définit le nom de la colonne où les valeurs de `String` seront réellement stockées.

Considérons le schéma mis à jour :



Vous pouvez voir que la clé primaire de la table de la collection est en fait une clé composée, utilisant les deux colonnes. Ceci implique aussi qu'il ne peut pas y avoir d'adresses email dupliquées par personne, ce qui est exactement la sémantique dont nous avons besoin pour un ensemble dans Java.

Vous pouvez maintenant tester et ajouter des éléments à cette collection, juste comme nous l'avons fait auparavant en liant des personnes et des événements. C'est le même code dans Java.

```
private void addEmailToPerson(Long personId, String emailAddress) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    // adding to the emailAddress collection might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

Cette fois-ci, nous n'avons pas utilisé de requête de chargement *fetch* pour initialiser la collection. Traquez les logs SQL et tentez d'optimiser ce cas avec un chargement agressif.

1.2.5. Associations bidirectionnelles

Ensuite nous allons mapper une association bidirectionnelle - faire fonctionner l'association entre une personne et un événement à partir des deux côtés dans Java. Bien sûr, le schéma de la base de données ne change pas, nous avons toujours une pluralité plusieurs-à-plusieurs.



Note

Une base de données relationnelle est plus flexible qu'un langage de programmation réseau, donc elle n'a pas besoin de direction de navigation - les données peuvent être vues et récupérées de toutes les manières possibles.

D'abord, ajoutez une collection de participants à la classe `Event` :

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

Maintenant mappez ce côté de l'association aussi, dans `Event.hbm.xml`.


```

<set name="participants" table="PERSON_EVENT" inverse="true">
  <key column="EVENT_ID" />
  <many-to-many column="PERSON_ID" class="events.Person" />
</set>
>

```

Comme vous le voyez, ce sont des mappages de `sets` normaux dans les deux documents de mappage. Notez que les noms de colonne dans `key` et `many-to-many` sont inversés dans les 2 documents de mappage. L'ajout le plus important ici est l'attribut `inverse="true"` dans l'élément `set` du mappage de la collection des `Events`.

Cela signifie que Hibernate devrait prendre l'autre côté - la classe `Person` - quand il a besoin de trouver des informations à propos du lien entre les deux. Ce sera beaucoup plus facile à comprendre une fois que vous verrez comment le lien bidirectionnel entre les deux entités est créé.

1.2.6. Travailler avec des liens bidirectionnels

Premièrement, gardez à l'esprit qu'Hibernate n'affecte pas la sémantique normale de Java. Comment avons-nous créé un lien entre une `Person` et un `Event` dans l'exemple unidirectionnel? Nous avons ajouté une instance de `Event` à la collection des références d'événement d'une instance de `Person`. Donc, évidemment, si vous voulons rendre ce lien bidirectionnel, nous devons faire la même chose de l'autre côté, en ajoutant une référence de `Person` à la collection dans un `Event`. Cette "configuration du lien des deux côtés" est absolument nécessaire et vous ne devriez jamais oublier de le faire.

Beaucoup de développeurs programment de manière défensive et créent des méthodes de gestion de lien pour affecter correctement les deux côtés, par exemple dans `Person` :

```

protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}

```

Notez que les méthodes `get` et `set` pour la collection sont maintenant protégées - ceci permet aux classes et aux sous-classes du même paquetage d'accéder aux méthodes, mais empêche quiconque de mettre le désordre directement dans les collections (enfin, presque). Vous devriez probablement faire de même avec la collection de l'autre côté.

Et à propos de l'attribut de mappage `inverse` ? Pour vous, et pour Java, un lien bidirectionnel consiste simplement à configurer correctement les références des deux côtés. Hibernate n'a cependant pas assez d'informations pour ordonner correctement les expressions SQL `INSERT` et `UPDATE` (pour éviter les violations de contrainte), et a besoin d'aide pour gérer proprement les associations bidirectionnelles. Rendre `inverse` un côté de l'association, indique à Hibernate de l'ignorer, pour le considérer comme un *miroir* de l'autre côté. Cela suffit à Hibernate pour gérer tous les problèmes de transformation d'un modèle de navigation directionnelle vers un schéma SQL de base de données. Les règles dont vous devez vous souvenir sont : toutes les associations bidirectionnelles ont besoin d'un côté marqué `inverse`. Dans une association un-à-plusieurs ce doit être le côté plusieurs, dans une association plusieurs-à-plusieurs, vous pouvez choisir n'importe quel côté, il n'y a pas de différence.

1.3. Section 3 - L'application web EventManager

Une application web Hibernate utilise la `Session` et `Transaction` comme une application autonome. Cependant, quelques modèles communs sont utiles. Nous allons coder une `EventManagerServlet`. Ce servlet peut lister tous les événements stockés dans la base de données, et fournir un formulaire HTML pour saisir de nouveaux événements.

1.3.1. Écrire la servlet de base

Tout d'abord, nous devons créer notre servlet de base. La servlet n'accepte que les requêtes HTTP `GET`, la méthode à implémenter est donc `doGet()` :

```
package org.hibernate.tutorial.web;

// Imports

public class EventManagerServlet extends HttpServlet {

    protected void doGet(
        HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

        SimpleDateFormat dateFormatter = new SimpleDateFormat( "dd.MM.yyyy" );

        try {
            // Begin unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().beginTransaction();

            // Process request and render page...

            // End unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit();
        }
    }
}
```

```

        catch (Exception ex) {
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().rollback();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }
}

```

Servir la servlet en tant que `src/main/java/org/hibernate/tutorial/web/EventManagerServlet.java`

Le modèle appliqué ici est appelé *session-per-request*. Lorsqu'une requête appelle la servlet, une nouvelle `Session` Hibernate est ouverte à la première invocation de `getCurrentSession()` sur la `SessionFactory`. Ensuite, une transaction avec la base de données est démarrée - tous les accès à la base de données interviennent au sein de la transaction, peu importe que les données soient lues ou écrites (nous n'utilisons pas le mode auto-commit dans les applications).

N'utilisez pas une nouvelle `Session` Hibernate pour chaque opération en base de données. Utilisez une `Session` Hibernate qui porte sur l'ensemble de la requête. Utilisez `getCurrentSession()`, ainsi elle est automatiquement attachée au thread Java courant.

Ensuite, les actions possibles de la requêtes sont exécutées et la réponse HTML est rendue. Nous y reviendrons ultérieurement.

Enfin, l'unité de travail s'achève lorsque l'exécution et le rendu sont achevés. Si un problème survient lors de ces deux phases, une exception est lancée et la transaction avec la base de données subit un rollback. Cela complète le modèle *session-per-request*. Au lieu d'avoir un code de délimitant les transactions au sein de chaque servlet, vous pouvez écrire un filtre de servlet. Voir le site Hibernate et le Wiki pour plus d'informations sur ce modèle, appelé *Open Session in View* - vous en aurez besoin dès que vous utiliserez des JSP et non des servlets pour le rendu de vos vues.

1.3.2. Traiter et interpréter

Implémentons l'exécution de la requête et le rendu de la page.

```

// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html

><head
><title
>Event Manager</title
></head
><body
>");

```

```
// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b");
    }
    else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
        out.println("<b");
    }

    ><i
    >Please enter event title and date.</i
    ></b
    >");
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body");
></html
>");

out.flush();
out.close();
```

Ce style de code avec une mixture de Java et d'HTML ne serait pas extensible dans une application plus complexe - gardez à l'esprit que nous ne faisons qu'illustrer les concepts basiques de Hibernate dans ce didacticiel. Ce code affiche une entête et un pied de page HTML. Dans cette page, sont affichés un formulaire pour la saisie d'évènements ainsi qu'une liste de tous les évènements de la base de données. La première méthode est triviale et ne fait que sortir de l'HTML :

```
private void printEventForm(PrintWriter out) {
    out.println("<h2");
    >Add new event:</h2
    >");

    out.println("<form");
    >");

    out.println("Title: <input name='eventTitle' length='50'/><br/>");
    out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
    out.println("<input type='submit' name='action' value='store'/>");
    out.println("</form");
    >");
}
```

La méthode `listEvents()` utilise la Session Hibernate liée au thread courant pour exécuter la requête :

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
        .getCurrentSession().createCriteria(Event.class).list();
    if (result.size()
> 0) {
        out.println("<h2
>Events in database:</h2
>");
        out.println("<table border='1'
>");
        out.println("<tr
>");
        out.println("<th
>Event title</th
>");
        out.println("<th
>Event date</th
>");
        out.println("</tr
>");
        Iterator it = result.iterator();
        while (it.hasNext()) {
            Event event = (Event) it.next();
            out.println("<tr
>");
            out.println("<td
>" + event.getTitle() + "</td
>");
            out.println("<td
>" + dateFormatter.format(event.getDate()) + "</td
>");
            out.println("</tr
>");
        }
        out.println("</table
>");
    }
}
```

Enfin, l'action `store` renvoie à la méthode `createAndStoreEvent()`, qui utilise aussi la Session du thread courant:

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
        .getCurrentSession().save(theEvent);
}
```

```
}
```

La servlet est complétée. Une requête à la servlet sera exécutée par une seule `Session` et `Transaction`. Comme dans l'application autonome vue auparavant, Hibernate peut automatiquement lier ces objets au thread courant d'exécution. Cela vous laisse la liberté de séparer votre code en couches et d'accéder à la `SessionFactory` selon le moyen que vous aurez choisi. Généralement, vous utiliserez des conceptions plus sophistiquées et déplacerez le code d'accès aux données dans une couche DAO. Consultez le wiki Hibernate pour plus d'exemples.

1.3.3. Déployer et tester

Pour déployer cette application en vue de procéder à des tests, nous devons créer un WAR (Web ARchive). Tout d'abord, nous devons définir le descripteur WAR en tant que `src/main/webapp/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

  <servlet>
    <servlet-name
>Event Manager</servlet-name>
    <servlet-class
>org.hibernate.tutorial.web.EventManagerServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name
>Event Manager</servlet-name>
    <url-pattern
>/eventmanager</url-pattern>
  </servlet-mapping>
</web-app>
>
```

Pour construire et déployer, appelez `ant war` dans votre projet et copiez le fichier `hibernate-tutorial.war` dans le répertoire `webapp` de Tomcat.



Note

If you do not have Tomcat installed, download it from <http://tomcat.apache.org/> and follow the installation instructions. Our application requires no changes to the standard Tomcat configuration.

Une fois l'application déployée et Tomcat lancé, accédez à l'application via `http://localhost:8080/hibernate-tutorial/eventmanager`. Assurez vous de consulter les traces Tomcat pour observer l'initialisation d'Hibernate à la première requête touchant votre servlet (l'initialisation statique dans `HibernateUtil` est invoquée) et pour vérifier qu'aucune exception ne survienne.

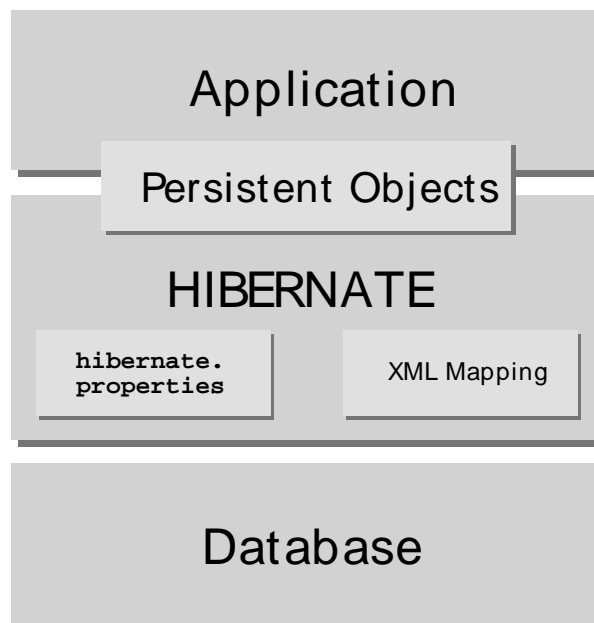
1.4. Résumé

Ce didacticiel a couvert les bases de l'écriture d'une simple application Hibernate ainsi qu'une petite application web. Vous trouverez des tutoriels supplémentaires dans le site Hibernate [website](http://hibernate.org) [<http://hibernate.org>].

Architecture

2.1. Généralités

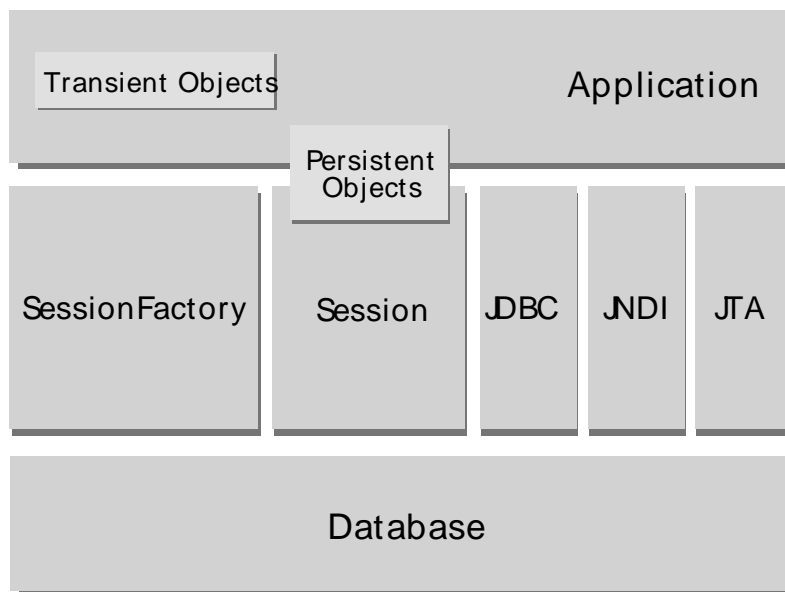
Le diagramme ci-dessus procure une vue - (très) haut niveau - de l'architecture Hibernate :



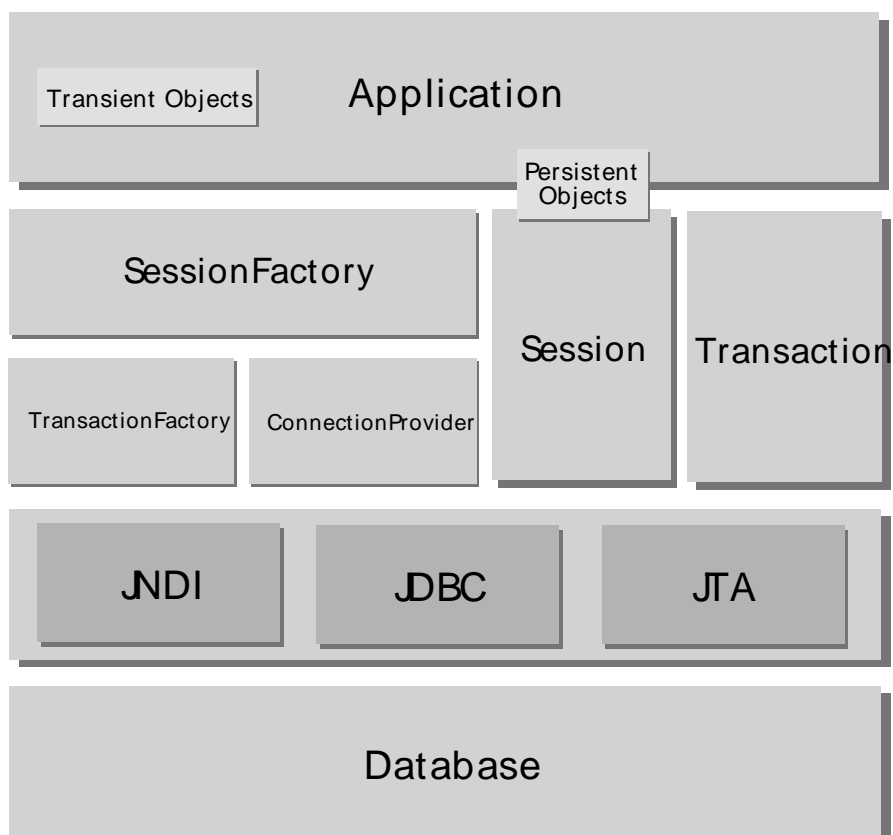
Nous aimerions décrire une vue plus détaillée de l'architecture. Hibernate est flexible et prend en charge différentes approches. Nous allons en montrer les deux extrêmes : l'architecture "légère" et l'architecture "complète".

Ce diagramme montre Hibernate utilisant la base de données et des données de configuration pour fournir un service de persistance, et des objets persistants, à l'application.

L'architecture "légère" permet à l'application de fournir ses propres connexions JDBC et de gérer ses propres transactions. Cette approche utilise un sous-ensemble minimum des API Hibernate :



L'architecture "complète" abstrait l'application des API JDBC/JTA sous-jacentes et permet à Hibernate de s'occuper des détails.



Voici quelques définitions des objets dans les diagrammes :

SessionFactory (`org.hibernate.SessionFactory`)

Un cache threadsafe (immuable) de mappages compilés pour une base de données. En tant que fabrique de `Session` et que client du `ConnectionProvider`, `SessionFactory` peut contenir un cache optionnel de données (de second niveau), réutilisable entre les différentes transactions, que cela soit au sein du même processus ou au niveau d'un cluster.

Session (`org.hibernate.Session`)

Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Fabrique des objets `Transaction`. La `Session` contient un cache (de premier niveau) des objets persistants, qui sont utilisés lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets et collections persistants

Objets mono-threadés à vie courte, contenant état persistant et fonction commerciale. Ceux-ci sont en général des objets ordinaires de type `JavaBean` (ou `POJO`); la seule particularité est qu'ils sont associés avec une (et une seule) `Session`. Dès que la `Session` est fermée, ils sont détachés et libres d'être utilisés par n'importe quelle couche de l'application (par ex. de et vers la présentation).

Objets et collections éphémères (transient) et détachés

Instances de classes persistantes qui ne sont actuellement pas associées à une `Session`. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées, ou elle ont pu être instanciées par une `Session` fermée.

Transaction (`org.hibernate.Transaction`)

(Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique. Abstrait l'application des transactions sous-jacentes, qu'elles soient JDBC, JTA ou CORBA. Une `Session` peut fournir plusieurs `Transactions` dans certains cas. Toutefois, la délimitation des transactions, via l'API d'Hibernate ou par la `Transaction` sous-jacente, n'est jamais optionnelle.

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)

(Optionnel) Une fabrique de (pool de) connexions JDBC. Abstrait l'application de la `Datasource` ou du `DriverManager` sous-jacent. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

TransactionFactory (`org.hibernate.TransactionFactory`)

(Optionnel) Une fabrique d'instances de `Transaction`. Non exposée à l'application, mais peut être étendue/implémentée par le développeur.

Extension Interfaces

Hibernate fournit de nombreuses interfaces d'extensions optionnelles que vous pouvez implémenter pour personnaliser le comportement de votre couche de persistance. Reportez vous à la documentation de l'API pour plus de détails.

Dans une architecture légère, l'application n'aura pas à utiliser les API `Transaction/TransactionFactory` et/ou n'utilisera pas les API `ConnectionProvider` pour utiliser directement JTA ou JDBC.

2.2. Etats des instances

Une instance d'une classe persistante peut être dans l'un des trois états suivants, définis par rapport à un *contexte de persistance*. L'objet `Session` Hibernate correspond à ce contexte de persistance. Les trois états distincts sont:

éphémère (transient)

L'instance n'est pas et n'a jamais été associée à un contexte de persistance. Elle ne possède pas d'identité persistante (valeur de clé primaire).

persistant

L'instance est associée à un contexte de persistance. Elle possède une identité persistante (valeur de clé primaire) et, peut-être un enregistrement correspondant dans la base de données. Pour un contexte de persistance particulier, Hibernate *garantit* que l'identité persistante soit équivalente à l'identité Java (emplacement mémoire de l'objet).

détaché

L'instance a été associée au contexte de persistance mais ce contexte a été fermé, ou l'instance a été sérialisée vers un autre processus. Elle possède une identité persistante et peut-être un enregistrement correspondant dans la base de données. Pour des instances détachées, Hibernate ne donne aucune garantie sur la relation entre l'identité persistante et l'identité Java.

2.3. Intégration JMX

JMX est le standard J2EE de gestion des composants Java. Hibernate peut être géré via un service JMX standard. Nous fournissons une implémentation d'un MBean dans la distribution : `org.hibernate.jmx.HibernateService`.

Pour un exemple sur la manière de déployer Hibernate en tant que service JMX dans le serveur d'application JBoss Application Server, référez vous au guide de l'utilisateur JBoss (JBoss User Guide). Si vous déployez Hibernate via JMX sur JBoss AS, vous aurez également les avantages suivants :

- *Gestion de la session* : le cycle de vie de la `Session` Hibernate peut être automatiquement liée à la portée d'une transaction JTA. Cela signifie que vous n'avez plus besoin d'ouvrir et de fermer la `Session` manuellement, cela devient le travail de l'intercepteur EJB de JBoss. Vous n'avez pas non plus à vous occuper des démarcations des transactions dans votre code (sauf si vous voulez écrire une couche de persistance qui soit portable, dans ce cas vous pouvez utiliser l'API optionnelle `Transaction` de Hibernate). Vous appelez le `HibernateContext` pour accéder à la `Session`.

- *Déploiement HAR* :: habituellement vous déployez le service JMX Hibernate en utilisant le descripteur de déploiement de JBoss dans un fichier EAR et/ou un SAR, car il supporte toutes les options de configuration usuelles d'une `SessionFactory` Hibernate. Cependant, vous devez toujours nommer tous vos fichiers de mappage dans le descripteur de déploiement. Si vous décidez d'utiliser le déploiement optionnel sous forme de HAR, JBoss détectera automatiquement tous vos fichiers de mapping dans votre fichier HAR.

Consultez le guide d'utilisation de JBoss AS pour plus d'informations sur ces options.

Another feature available as a JMX service is runtime Hibernate statistics. See [Section 3.4.6](#), « *Statistiques Hibernate* » for more information.

2.4. Support JCA

Hibernate peut aussi être configuré en tant que connecteur JCA. Référez-vous au site web pour de plus amples détails. Il est important de noter que le support JCA de Hibernate est encore considéré comme expérimental.

2.5. Sessions contextuelles

Certaines applications utilisant Hibernate ont besoin d'une sorte de session "contextuelle", où une session donnée est en effet liée à la portée d'un contexte particulier. Cependant, les applications ne définissent pas toutes la notion de contexte de la même manière, et différents contextes définissent différentes portées à la notion de "courant". Les applications qui utilisaient Hibernate, versions précédentes à la 3.0, avaient tendance à employer un principe maison de sessions contextuelles basées sur le `ThreadLocal`, ainsi que sur des classes utilitaires comme `HibernateUtil`, ou utilisaient des framework tiers (comme Spring ou Pico) qui fournissaient des sessions contextuelles basées sur l'utilisation de proxy/interception.

A partir de la version 3.0.1, Hibernate a ajouté la méthode `SessionFactory.getCurrentSession()`. Initialement, cela demandait l'usage de transactions JTA, où la transaction JTA définissait la portée et le contexte de la session courante. L'équipe Hibernate pense que, étant donnée la maturité des nombreuses implémentations autonomes du JTA `TransactionManager`, la plupart (sinon toutes) des applications devraient utiliser la gestion des transactions par JTA qu'elles soient ou non déployées dans un conteneur J2EE. Par conséquent, il vous suffira de contextualiser vos sessions via la méthode basée sur JTA.

Cependant, depuis la version 3.1, la logique derrière `SessionFactory.getCurrentSession()` est désormais enfichable. A cette fin, une nouvelle interface d'extension (`org.hibernate.context.CurrentSessionContext`) et un nouveau paramètre de configuration `hibernate.current_session_context_class` ont été ajoutés pour enficher la portée et le contexte de sessions courantes caractéristiques.

Pour une description détaillée de son contrat, consultez les Javadocs de l'interface `org.hibernate.context.CurrentSessionContext`. Elle définit une seule méthode, `currentSession()`, par laquelle l'implémentation est responsable de traquer la session contextuelle courante. Hibernate fournit trois implémentations de cette interface :

- `org.hibernate.context.JTASessionContext` - les sessions courantes sont associées à une transaction JTA. La logique est la même que l'ancienne approche basée sur JTA. Consultez les javadocs pour plus d'informations.
- `org.hibernate.context.ThreadLocalSessionContext` - les sessions courantes sont traquées par l'exécution du thread. Consultez les javadocs pour plus d'informations.
- `org.hibernate.context.ManagedSessionContext` - les sessions courantes sont traquées par l'exécution du thread. Toutefois, vous êtes responsable de lier et de délier une instance de `Session` avec des méthodes statiques de cette classe. Elle n'ouvre jamais, ni ne nettoie ou ne ferme une `Session`.

The first two implementations provide a "one session - one database transaction" programming model. This is also known and used as *session-per-request*. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain JSE without JTA, you are advised to use the Hibernate `Transaction` API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. Refer to [Chapitre 12, Transactions et Accès concurrents](#) for more information and code examples.

Le paramètre de configuration `hibernate.current_session_context_class` définit quelle implémentation de `org.hibernate.context.CurrentSessionContext` doit être utilisée. Notez que pour assurer la compatibilité avec les versions précédentes, si ce paramètre n'est pas défini mais qu'un `org.hibernate.transaction.TransactionManagerLookup` est configuré, Hibernate utilisera le `org.hibernate.context.JTASessionContext`. La valeur de ce paramètre devrait juste nommer la classe d'implémentation à utiliser. Pour les trois implémentations prêtes à utiliser, toutefois, il y a trois noms brefs correspondants : "jta", "thread" et "managed".

Configuration

Hibernate est conçu pour fonctionner dans de nombreux environnements , c'est pourquoi il existe beaucoup de paramètres de configuration. Heureusement, la plupart ont des valeurs par défaut appropriées et la Hibernate inclut un fichier d'exemples `hibernate.properties` dans le répertoire `etc/` qui fournit les différentes options. Vous n'avez qu'à placer ce fichier dans votre classpath et à l'adapter à vos besoins.

3.1. Configuration par programmation

Une instance de `org.hibernate.cfg.Configuration` représente un ensemble de mappages des classes Java d'une application vers la base de données SQL. La `Configuration` est utilisée pour construire un objet (immuable) `SessionFactory`. Les mappages sont constitués d'un ensemble de fichiers de mappage XML.

Vous pouvez obtenir une instance de `Configuration` en l'instanciant directement et en spécifiant la liste des documents XML de mappage. Si les fichiers de mappage sont dans le classpath, vous pouvez utiliser la méthode `addResource()` :

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

Une solution alternative consiste à spécifier la classe mappée et à donner à Hibernate la possibilité de trouver les documents de mappage pour vous :

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate va rechercher les fichiers de mappages `/org/hibernate/auction/Item.hbm.xml` et `/org/hibernate/auction/Bid.hbm.xml` dans le classpath. Cette approche élimine les noms de fichiers en dur.

Une `Configuration` vous permet également de préciser des propriétés de configuration. Par exemple :

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

Ce n'est pas le seul moyen de passer des propriétés de configuration à Hibernate. Les différentes options sont :

1. Passer une instance de `java.util.Properties` à `Configuration.setProperties()`.
2. Placer `hibernate.properties` dans un répertoire racine du chemin de classe.
3. Positionner les propriétés System en utilisant `java -Dproperty=value`.
4. Inclure des éléments `<property>` dans le fichier `hibernate.cfg.xml` (voir plus loin).

Si vous souhaitez démarrer rapidement, `hibernate.properties` est l'approche la plus facile.

`org.hibernate.cfg.Configuration` est un objet de démarrage qui sera supprimé une fois qu'une `SessionFactory` aura été créée.

3.2. Obtenir une SessionFactory

When all mappings have been parsed by the `org.hibernate.cfg.Configuration`, the application must obtain a factory for `org.hibernate.Session` instances. This factory is intended to be shared by all application threads:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate does allow your application to instantiate more than one `org.hibernate.SessionFactory`. This is useful if you are using more than one database.

3.3. Connexions JDBC

Il est conseillé que `org.hibernate.SessionFactory` crée les connexions JDBC et les mette dans un pool pour vous. Si vous suivez cette approche, ouvrir une `org.hibernate.Session` est aussi simple que :

```
Session session = sessions.openSession(); // open a new Session
```

Dès que vous initiez une action qui requiert un accès à la base de données, une connexion JDBC sera récupérée dans le pool.

À cet effet, il faut passer les propriétés de la connexion JDBC à Hibernate. Tous les noms des propriétés Hibernate et leur signification sont définies dans la classe `org.hibernate.cfg.Environment`. Nous allons maintenant décrire les paramètres de configuration des connexions JDBC les plus importants.

Hibernate obtiendra des connexions (et les mettra dans un pool) en utilisant `java.sql.DriverManager` si vous positionnez les paramètres de la manière suivante :

Tableau 3.1. Propriétés JDBC de Hibernate

Nom de la propriété	Fonction
<code>hibernate.connection.driver_class</code>	<i>JDBC driver class</i>
<code>hibernate.connection.url</code>	<i>JDBC URL</i>
<code>hibernate.connection.username</code>	<i>database user</i>
<code>hibernate.connection.password</code>	<i>database user password</i>
<code>hibernate.connection.pool_size</code>	<i>maximum number of pooled connections</i>

L'algorithme natif de pool de connexions de Hibernate est plutôt rudimentaire. Il a été conçu dans le but de vous aider à démarrer et *n'est pas prévu pour un système en production* ou même pour un test de performance. Utilisez plutôt un pool tiers pour de meilleures performances et une meilleure stabilité : remplacez la propriété `hibernate.connection.pool_size` avec les propriétés spécifiques au pool de connexions que vous avez choisi. Cela désactivera le pool de connexions interne de Hibernate. Vous pouvez par exemple utiliser C3P0.

C3P0 est un pool de connexions JDBC open source distribué avec Hibernate dans le répertoire `lib`. Hibernate utilisera son provider `C3P0ConnectionProvider` pour le pool de connexions si vous configurez les propriétés `hibernate.c3p0.*`. Si vous voulez utiliser Proxool, référez vous au groupe de propriétés `hibernate.properties` correspondant et consultez le site web Hibernate pour plus d'informations.

Voici un exemple de fichier `hibernate.properties` pour C3P0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Pour l'utilisation de Hibernate au sein d'un serveur d'applications, il est recommandé de configurer Hibernate presque toujours de façon à ce qu'il obtienne ses connexions de la `DataSource` enregistrée du serveur d'applications dans le JNDI. À cet effet, vous devrez définir au moins une des propriétés suivantes :

Tableau 3.2. Propriétés d'une DataSource Hibernate

Nom de la propriété	Fonction
<code>hibernate.connection.datasource</code>	<i>datasource JNDI name</i>
<code>hibernate.jndi.url</code>	<i>URL du fournisseur JNDI (optionnel)</i>

Nom de la propriété	Fonction
hibernate.jndi.class	<i>classe de JNDI InitialContextFactory (optionnel)</i>
hibernate.connection.username	<i>utilisateur de base de données (optionnel)</i>
hibernate.connection.password	<i>mot de passe de l'utilisateur de base de données (optionnel)</i>

Voici un exemple de fichier `hibernate.properties` pour l'utilisation d'une datasource JNDI fournie par un serveur d'applications :

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Les connexions JDBC obtenues à partir d'une datasource JNDI participeront automatiquement aux transactions gérées par le conteneur du serveur d'applications.

Des propriétés arbitraires de connexion peuvent être passées en préfixant le nom de la propriété par `"hibernate.connection."`. Par exemple, vous pouvez spécifier un `charSet` en utilisant `hibernate.connection.charSet`.

Vous pouvez fournir votre propre stratégie d'obtention des connexions JDBC en implémentant l'interface `org.hibernate.connection.ConnectionProvider`. Vous pouvez sélectionner une implémentation spécifique par la propriété `hibernate.connection.provider_class`.

3.4. Propriétés de configuration optionnelles

Il y a un certain nombre d'autres propriétés qui contrôlent le fonctionnement d'Hibernate à l'exécution. Toutes sont optionnelles et ont comme valeurs par défaut des valeurs raisonnables.



Avertissement

*Some of these properties are "system-level" only. System-level properties can be set only via `java -Dproperty=value` or `hibernate.properties`. They *cannot* be set by the other techniques described above.*

Tableau 3.3. Propriétés de configuration Hibernate

Nom de la propriété	Fonction
hibernate.dialect	Le nom de la classe d'un <code>org.hibernate.dialect.Dialect</code> Hibernate

Nom de la propriété	Fonction
	<p>qui permet à Hibernate de générer du SQL optimisé pour une base de données relationnelle particulière.</p> <p>par ex. <code>full.classname.of.Dialect</code></p> <p>Dans la plupart des cas, Hibernate sera en mesure de choisir l'implémentation <code>org.hibernate.dialect.Dialect</code> qui convient sur la base des métadonnées JDBC retournées par le driver JDBC.</p>
<code>hibernate.show_sql</code>	<p>Écrit toutes les requêtes SQL sur la console. Il s'agit d'une alternative au paramétrage de la catégorie de log <code>org.hibernate.SQL</code> à debug.</p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.format_sql</code>	<p>Effectue un pretty print du SQL dans la console et dans le log.</p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.default_schema</code>	<p>Qualifie des noms de table non qualifiés avec le schéma/tablespace dans le SQL généré.</p> <p>e.g. <code>SCHEMA_NAME</code></p>
<code>hibernate.default_catalog</code>	<p>Qualifie les noms de tables non qualifiées avec ce catalogue dans le SQL généré.</p> <p>e.g. <code>CATALOG_NAME</code></p>
<code>hibernate.session_factory_name</code>	<p><code>org.hibernate.SessionFactory</code> sera automatiquement liée à ce nom dans JNDI après sa création.</p> <p>par ex. <code>jndi/composite/name</code></p>
<code>hibernate.max_fetch_depth</code>	<p>Configure la profondeur maximale d'un arbre de chargement par jointures externes pour les associations à cardinalité unitaire (un-à-un, plusieurs-à-un). Un 0 désactive le chargement par défaut par jointure externe.</p> <p>par ex. valeurs recommandées entre 0 et 3</p>
<code>hibernate.default_batch_fetch_size</code>	<p>Configure une taille par défaut pour le chargement par lot des associations Hibernate</p> <p>ex. valeurs recommandées : 4, 8, 16</p>

Nom de la propriété	Fonction
hibernate.default_entity_mode	Sets a default mode for entity representation for all sessions opened from this SessionFactory dynamic-map, dom4j, pojo
hibernate.order_updates	Force Hibernate à trier les mises à jour SQL par la valeur de la clé primaire des éléments mis à jour. Cela permet de limiter les deadlocks de transaction dans les systèmes hautement concurrents. par ex. true false
hibernate.generate_statistics	Si activé, Hibernate va collecter des statistiques utiles pour le réglage des performances. par ex. true false
hibernate.use_identifier_rollback	Si activé, les propriétés correspondant à l'identifiant des objets sont remises aux valeurs par défaut lorsque les objets sont supprimés. par ex. true false
hibernate.use_sql_comments	Si activé, Hibernate génère des commentaires à l'intérieur des requêtes SQL pour faciliter le débogage, par défaut à false. par ex. true false

Tableau 3.4. Propriétés Hibernate liées à JDBC et aux connexions

Nom de la propriété	Fonction
hibernate.jdbc.fetch_size	Une valeur non nulle détermine la taille des chargements JDBC (appelle Statement.setFetchSize()).
hibernate.jdbc.batch_size	Une valeur non nulle active l'utilisation par Hibernate des mise à jour par lot de JDBC2. ex. les valeurs recommandées entre 5 et 30
hibernate.jdbc.batch_versioned_data	Set this property to true if your JDBC driver returns correct row counts from executeBatch(). It is usually safe to turn this option on. Hibernate will then use batched DML

Nom de la propriété	Fonction
	<p>for automatically versioned data. Defaults to false.</p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.jdbc.factory_class</code>	<p>Sélectionne un <code>org.hibernate.jdbc.Batcher</code> personnalisé. La plupart des applications n'auront pas besoin de cette propriété de configuration.</p> <p>par ex. <code>classname.of.BatcherFactory</code></p>
<code>hibernate.jdbc.use_scrollable_resultset</code>	<p>Active l'utilisation par Hibernate des ensembles de résultats déroulants de JDBC2. Cette propriété est seulement nécessaire lorsque l'on utilise des connexions JDBC fournies par l'utilisateur. Autrement, Hibernate utilise les métadonnées de la connexion.</p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.jdbc.use_streams_for_binary</code>	<p>Utilise des flux lorsque l'on écrit/lit des types <code>binary</code> ou des types <code>serializable</code> vers/à partir de JDBC. <i>*system-level property*</i></p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.jdbc.use_get_generated_keys</code>	<p>Active l'utilisation de <code>PreparedStatement.getGeneratedKeys()</code> de JDBC3 pour récupérer nativement les clés générées après insertion. Nécessite un pilote JDBC3+ et JRE1.4+, configurés à <code>false</code> si votre pilote a des problèmes avec les générateurs d'identifiant Hibernate. Par défaut, essaie de déterminer les possibilités du pilote en utilisant les métadonnées de connexion.</p> <p>par ex. <code>true</code> <code>false</code></p>
<code>hibernate.connection.provider_class</code>	<p>Le nom de la classe d'un <code>org.hibernate.connection.ConnectionProvider</code> personnalisé qui fournit des connexions JDBC à Hibernate.</p> <p>par ex. <code>classname.of.ConnectionProvider</code></p>
<code>hibernate.connection.isolation</code>	<p>Définit le niveau d'isolation des transactions JDBC. Regardez <code>java.sql.Connection</code> pour des valeurs significatives mais notez</p>

Nom de la propriété	Fonction
	<p>également que la plupart des bases de données ne supportent pas tous les niveaux d'isolation et que certaines définissent des isolations non standard supplémentaires.</p> <p>par ex. 1, 2, 4, 8</p>
hibernate.connection.autocommit	<p>Active le mode de commit automatique (autocommit) pour les connexions JDBC du pool (non recommandé).</p> <p>par ex. true false</p>
hibernate.connection.release_mode	<p>Spécifie à quel moment Hibernate doit relâcher les connexions JDBC. Par défaut, une connexion JDBC est conservée jusqu'à ce que la session soit explicitement fermée ou déconnectée. Pour une source de données JTA d'un serveur d'applications, vous devriez utiliser <code>after_statement</code> pour libérer les connexions de manière plus agressive après chaque appel JDBC. Pour une connexion non JTA, il est souvent préférable de libérer la connexion à la fin de chaque transaction en utilisant <code>after_transaction</code>. <code>auto</code> choisira <code>after_statement</code> pour les stratégies de transactions JTA et CMT et <code>after_transaction</code> pour des stratégies de transactions JDBC.</p> <p>e.g. auto (default) on_close after_transaction after_statement</p> <p>This setting only affects Sessions returned from <code>SessionFactory.openSession()</code>. For Sessions obtained through <code>SessionFactory.getCurrentSession()</code>, the <code>CurrentSessionContext</code> implementation configured for use controls the connection release mode for those Sessions. See Section 2.5, « Sessions contextuelles »</p>
hibernate.connection.<propertyName>	<p>Passez une propriété JDBC <code>propertyName</code> à <code>DriverManager.getConnection()</code>.</p>
hibernate.jndi.<propertyName>	<p>Passez la propriété <propertyName> au JNDI <code>InitialContextFactory</code>.</p>

Tableau 3.5. Propriétés du Cache Hibernate

Nom de la propriété	Fonction
<code>hibernate.cache.provider_class</code>	<p>Le nom de classe d'un <code>CacheProvider</code> personnalisé.</p> <p>par ex. <code>classname.of.CacheProvider</code></p>
<code>hibernate.cache.use_minimal_puts</code>	<p>Optimise le cache de second niveau en minimisant les écritures, au prix de plus de lectures. Ce paramètre est surtout utile pour les caches en cluster, et est activé par défaut dans hibernate3 pour les implémentations de cache en cluster.</p> <p>par ex. <code>true false</code></p>
<code>hibernate.cache.use_query_cache</code>	<p>Activer le cache de requête, les requêtes individuelles doivent tout de même être déclarées comme pouvant être mises en cache.</p> <p>par ex. <code>true false</code></p>
<code>hibernate.cache.use_second_level_cache</code>	<p>Peut être utilisé pour désactiver complètement le cache de second niveau qui est activé par défaut pour les classes qui spécifient un élément <code><cache></code> dans leur mappage.</p> <p>par ex. <code>true false</code></p>
<code>hibernate.cache.query_cache_factory</code>	<p>Le nom de classe d'une interface <code>QueryCache</code> personnalisée, par défaut prend la valeur du <code>StandardQueryCache</code> imbriqué.</p> <p>par ex. <code>classname.of.QueryCache</code></p>
<code>hibernate.cache.region_prefix</code>	<p>Un préfixe à utiliser pour les noms de régions du cache de second niveau.</p> <p>par ex. <code>prefix</code></p>
<code>hibernate.cache.use_structured_entries</code>	<p>Force Hibernate à stocker les données dans le cache de second niveau en un format plus adapté à la visualisation.</p> <p>par ex. <code>true false</code></p>

Tableau 3.6. Propriétés des transactions Hibernate

Nom de la propriété	Fonction
<code>hibernate.transaction.factory_class</code>	<p>Le nom de classe d'une <code>TransactionFactory</code> qui sera utilisée par l'API Transaction de Hibernate (la valeur par défaut est <code>JDBCTransactionFactory</code>).</p> <p>par ex. <code>classname.of.TransactionFactory</code></p>
<code>jta.UserTransaction</code>	<p>Le nom JNDI utilisé par la <code>JATransactionFactory</code> pour obtenir la <code>UserTransaction</code> JTA du serveur d'applications.</p> <p>par ex. <code>jndi/composite/name</code></p>
<code>hibernate.transaction.manager_lookup_class</code>	<p>Le nom de la classe d'une <code>TransactionManagerLookup</code> - requise lorsque le cache de niveau JVM est activé ou lorsque l'on utilise un générateur hilo dans un environnement JTA.</p> <p>par ex. <code>classname.of.TransactionManagerLookup</code></p>
<code>hibernate.transaction.flush_before_completion</code>	<p>If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see Section 2.5, « Sessions contextuelles ».</p> <p>par ex. <code>true false</code></p>
<code>hibernate.transaction.auto_close_session</code>	<p>If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see Section 2.5, « Sessions contextuelles ».</p> <p>par ex. <code>true false</code></p>

Tableau 3.7. Propriétés diverses

Nom de la propriété	Fonction
<code>hibernate.current_session_context_class</code>	<p>Supply a custom strategy for the scoping of the "current" Session. See Section 2.5, « Sessions contextuelles » for more information about the built-in strategies.</p>

Nom de la propriété	Fonction
	e.g. jta thread managed custom.Class
hibernate.query.factory_class	Choisit l'implémentation du parseur de requête HQL. par ex. org.hibernate.hql.ast.ASTQueryTranslatorFactory ou org.hibernate.hql.classic.ClassicQueryTranslatorFactory
hibernate.query.substitutions	Lien entre les jetons de requêtes Hibernate et les jetons SQL (les jetons peuvent être des fonctions ou des noms textuels par exemple). par ex. hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC
hibernate.hbm2ddl.auto	Valide ou exporte automatiquement le schéma DDL vers la base de données lorsque la SessionFactory est créée. La valeur create-drop permet de supprimer le schéma de base de données lorsque la SessionFactory est fermée explicitement. par ex. validate update create create-drop
hibernate.bytecode.use_reflection_optimizer	Enables the use of bytecode manipulation instead of runtime reflection. This is a System-level property and cannot be set in hibernate.cfg.xml. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either CGLIB or javassist even if you turn off the optimizer. par ex. true false
hibernate.bytecode.provider	Both javassist or cglib can be used as byte manipulation engines; the default is javassist. e.g. javassist cglib

3.4.1. Dialectes SQL

Il est recommandé de toujours positionner la propriété `hibernate.dialect` à la sous-classe de `org.hibernate.dialect.Dialect` appropriée à votre base de données. Si vous spécifiez un dialecte, Hibernate utilisera des valeurs adaptées pour certaines autres propriétés listées ci-dessus, vous évitant ainsi de l'effectuer à la main.

Tableau 3.8. Dialectes SQL de Hibernate (`hibernate.dialect`)

RDBMS	Dialecte
DB2	<code>org.hibernate.dialect.DB2Dialect</code>
DB2 AS/400	<code>org.hibernate.dialect.DB2400Dialect</code>
DB2 OS390	<code>org.hibernate.dialect.DB2390Dialect</code>
PostgreSQL	<code>org.hibernate.dialect.PostgreSQLDialect</code>
MySQL	<code>org.hibernate.dialect.MySQLDialect</code>
MySQL with InnoDB	<code>org.hibernate.dialect.MySQLInnoDBDialect</code>
MySQL with MyISAM	<code>org.hibernate.dialect.MySQLMyISAMDialect</code>
Oracle (toutes versions)	<code>org.hibernate.dialect.OracleDialect</code>
Oracle 9i	<code>org.hibernate.dialect.Oracle9iDialect</code>
Oracle 10g	<code>org.hibernate.dialect.Oracle10gDialect</code>
Sybase	<code>org.hibernate.dialect.SybaseDialect</code>
Sybase Anywhere	<code>org.hibernate.dialect.SybaseAnywhereDialect</code>
Microsoft SQL Server	<code>org.hibernate.dialect.SQLServerDialect</code>
SAP DB	<code>org.hibernate.dialect.SAPDBDialect</code>
Informix	<code>org.hibernate.dialect.InformixDialect</code>
HypersonicSQL	<code>org.hibernate.dialect.HSQLDialect</code>
Ingres	<code>org.hibernate.dialect.IngresDialect</code>
Progress	<code>org.hibernate.dialect.ProgressDialect</code>
Mckoi SQL	<code>org.hibernate.dialect.MckoiDialect</code>
Interbase	<code>org.hibernate.dialect.InterbaseDialect</code>
Pointbase	<code>org.hibernate.dialect.PointbaseDialect</code>
FrontBase	<code>org.hibernate.dialect.FrontbaseDialect</code>
Firebird	<code>org.hibernate.dialect.FirebirdDialect</code>

3.4.2. Chargement par jointure externe

Si votre base de données supporte les jointures externes de type ANSI, Oracle ou Sybase, *le chargement par jointure externe* devrait améliorer les performances en limitant le nombre d'allers-retours avec la base de données (la base de données effectuant donc potentiellement plus de travail). Le chargement par jointure ouverte permet à un graphe entier d'objets connectés par une relation plusieurs-à-un, un-à-plusieurs ou un-à-un d'être chargé en un seul `SQLSELECT`.

Le chargement par jointure ouverte peut être désactivé *globalement* en mettant la propriété `hibernate.max_fetch_depth` à 0. Une valeur de 1 ou plus active le chargement par jointure externe pour les associations un-à-un et plusieurs-à-un qui ont été mappées avec `fetch="join"`.

See [Section 20.1, « Stratégies de chargement »](#) for more information.

3.4.3. Flux binaires

Oracle limite la taille d'un tableau d'octets qui peuvent être passés vers et à partir de son pilote JDBC. Si vous souhaitez utiliser des instances larges de type `binary` ou `serializable`, vous devez activer la propriété `hibernate.jdbc.use_streams_for_binary`. *C'est une fonctionnalité de niveau système uniquement.*

3.4.4. Cache de second niveau et cache de requêtes

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the [Section 20.2, « Le cache de second niveau »](#) for more information.

3.4.5. Substitution dans le langage de requêtes

Vous pouvez définir de nouveaux jetons dans les requêtes Hibernate en utilisant `hibernate.query.substitutions`. Par exemple :

```
hibernate.query.substitutions true=1, false=0
```

Cela signifierait que les jetons `true` et `false` seraient transformés par des entiers dans le SQL généré.

```
hibernate.query.substitutions toLowercase=LOWER
```

Cela permettrait de renommer la fonction SQL `LOWER`.

3.4.6. Statistiques Hibernate

Si vous activez `hibernate.generate_statistics`, Hibernate fournira un certain nombre de métriques utiles pour régler les performances d'une application qui tourne via `SessionFactory.getStatistics()`. Hibernate peut aussi être configuré pour exposer ces statistiques via JMX. Lisez les Javadoc des interfaces dans le paquetage `org.hibernate.stats` pour plus d'informations.

3.5. Journalisation

Hibernate utilise [Simple Logging Facade for Java](http://www.slf4j.org/) [http://www.slf4j.org/] (SLF4J) pour enregistrer divers événements du système. SLF4J peut diriger votre sortie de logging vers plusieurs structures de loggings (NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL or logback) suivant la liaison que vous choisirez. Pour pouvoir configurer votre logging, vous aurez besoin de `slf4j-api.jar` dans votre chemin de classe, ainsi que du fichier jar pour votre liaison préférée - `slf4j-log4j12.jar` pour Log4J. Voir la documentation SLF4J [documentation](http://www.slf4j.org/manual.html) [http://www.slf4j.org/manual.html] pour davantage d'informations. Pour utiliser Log4j, vous aurez aussi besoin de

mettre un fichier `log4j.properties` dans votre chemin de classe. Un exemple de fichier de propriétés est distribué avec Hibernate dans le répertoire `src/`.

Il est vivement recommandé de vous familiariser avec les messages des logs de Hibernate. Beaucoup de soin a été apporté pour donner le plus de détails possible sans les rendre illisibles. C'est un outil essentiel en cas de problèmes. Les catégories de logs les plus intéressantes sont les suivantes :

Tableau 3.9. Catégories de logs de Hibernate

Catégorie	Fonction
<code>org.hibernate.SQL</code>	Journalise toutes les requêtes SQL de type DML (gestion des données) qui sont exécutées
<code>org.hibernate.type</code>	Journalise tous les paramètres JDBC
<code>org.hibernate.tool.hbm2ddl</code>	Journalise toutes les requêtes SQL de type DDL (gestion de la structure de la base) qui sont exécutées
<code>org.hibernate.pretty</code>	Journalise l'état de toutes les entités (20 entités maximum) associées avec la session Hibernate au moment du flush
<code>org.hibernate.cache</code>	Journalise toute activité du cache de second niveau
<code>org.hibernate.transaction</code>	Journalise toute activité relative aux transactions
<code>org.hibernate.jdbc</code>	Journalise toute acquisition de ressource JDBC
<code>org.hibernate.hql.ast.AST</code>	Journalise l'arbre syntaxique des requêtes HQL et SQL durant l'analyse syntaxique des requêtes
<code>org.hibernate.secure</code>	Journalise toutes les demandes d'autorisation JAAS
<code>org.hibernate</code>	Journalise tout (beaucoup d'informations, mais très utile pour résoudre les problèmes).

Lorsque vous développez des applications avec Hibernate, vous devriez quasiment toujours travailler avec le niveau `debug` activé pour la catégorie `org.hibernate.SQL`, ou sinon avec la propriété `hibernate.show_sql` activée.

3.6. Sélectionne une `NamingStrategy` (stratégie de nommage)

L'interface `org.hibernate.cfg.NamingStrategy` vous permet de spécifier une "stratégie de nommage" des objets et éléments de la base de données.

Vous pouvez fournir des règles pour automatiquement générer les identifiants de base de données à partir des identifiants Java, ou transformer une colonne ou table "logique" donnée dans le fichier de mappage en une colonne ou table "physique". Cette fonctionnalité aide à réduire la verbosité de documents de mappage, en éliminant le bruit répétitif (les préfixes `TBL_` par exemple). La stratégie par défaut utilisée par Hibernate est assez minimale.

Vous pouvez définir une stratégie différente en appelant `Configuration.setNamingStrategy()` avant d'ajouter des mappages :

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`net.sf.hibernate.cfg.ImprovedNamingStrategy` est une stratégie fournie qui peut être utile comme point de départ de quelques applications.

3.7. Fichier de configuration XML

Une approche alternative est de spécifier toute la configuration dans un fichier nommé `hibernate.cfg.xml`. Ce fichier peut être utilisé à la place du fichier `hibernate.properties`, voire même peut servir à surcharger les propriétés si les deux fichiers sont présents.

Le fichier de configuration XML doit par défaut se placer à la racine du `CLASSPATH`. En voici un exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource"
>java:/comp/env/jdbc/MyDB</property>
        <property name="dialect"
>org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql"
>false</property>
        <property name="transaction.factory_class"
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction"
>java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>
```

```
</hibernate-configuration>  
>
```

Comme vous pouvez le constater, l'avantage de cette approche est l'externalisation des noms des fichiers de mappage de la configuration. Le fichier `hibernate.cfg.xml` est également plus pratique quand on commence à régler le cache d'Hibernate. Notez que vous pouvez choisir entre utiliser `hibernate.properties` ou `hibernate.cfg.xml`, les deux sont équivalents, sauf en ce qui concerne les bénéfices de l'utilisation de la syntaxe XML mentionnés ci-dessus.

Avec la configuration XML, démarrer Hibernate devient donc aussi simple que ceci :

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Vous pouvez choisir un fichier de configuration XML différent en utilisant :

```
SessionFactory sf = new Configuration()  
    .configure("catdb.cfg.xml")  
    .buildSessionFactory();
```

3.8. Intégration à un serveur d'applications J2EE

Hibernate possède les points d'intégration suivants pour l'infrastructure J2EE :

- *Source de données gérée par le conteneur* : Hibernate peut utiliser des connexions JDBC gérées par le conteneur et fournies par l'intermédiaire de JNDI. Souvent, un `TransactionManager` compatible JTA et un `ResourceManager` s'occupent de la gestion des transactions (CMT). Ils sont conçus en particulier pour gérer des transactions distribuées sur plusieurs sources de données. Vous pouvez bien sûr également définir les limites des transactions dans votre programme (BMT) ou vous pouvez par ailleurs utiliser l'API optionnelle `Transaction` de Hibernate qui vous garantira la portabilité de votre code entre plusieurs serveurs d'application.
- *Association JNDI automatique*: Hibernate peut associer sa `SessionFactory` à JNDI après le démarrage.
- *Association de la Session à JTA*: la `Session` Hibernate peut être automatiquement associée à la portée des transactions JTA si vous utilisez les EJB. Vous avez juste à récupérer la `SessionFactory` depuis JNDI et à récupérer la `Session` courante. Hibernate s'occupe de vider et fermer la `Session` lorsque votre transaction JTA se termine. La démarcation des transactions se fait de manière déclarative (CMT) ou de façon programmatique (BMT/UserTransaction).
- *Déploiement JMX* : Si vous avez un serveur d'applications compatible JMX (JBoss AS par exemple), vous pouvez choisir de déployer Hibernate en tant que MBean géré par le serveur. Cela vous évite de coder la ligne de démarrage qui permet de construire la `SessionFactory`

depuis la `Configuration`. Le conteneur va démarrer votre `HibernateService`, et va idéalement s'occuper des dépendances entre les services (la source de données doit être disponible avant le démarrage de Hibernate, etc).

En fonction de votre environnement, vous mettrez l'option de configuration `hibernate.connection.aggressive_release` à `true` si le serveur d'applications affiche des exceptions de type "connection containment".

3.8.1. Configuration de la stratégie transactionnelle

L'API de la `Session` Hibernate est indépendante de tout système de démarcation des transactions, présent dans votre architecture. Si vous laissez Hibernate utiliser l'API JDBC directement via un pool de connexion, vous commencerez et terminerez vos transactions en appelant l'API JDBC. Si votre application tourne à l'intérieur d'un serveur d'applications J2EE, vous utiliserez peut être les transactions gérées par les beans (BMT) et vous appellerez l'API JTA et `UserTransaction` lorsque cela est nécessaire.

Pour conserver votre code portable entre ces deux environnements (et d'autres), nous vous recommandons d'utiliser l'API optionnelle `Transaction` d'Hibernate, qui encapsule et masque le système de transaction sous-jacent. Pour cela, vous devez préciser une classe de fabrique d'instances de `Transaction` en positionnant la propriété de configuration `hibernate.transaction.factory_class`.

Il existe trois choix standards (intégrés) :

`org.hibernate.transaction.JDBCTransactionFactory`

délègue aux transactions de la base de données (JDBC) (valeur par défaut).

`org.hibernate.transaction.JTATransactionFactory`

délègue à CMT si une transaction existante est sous ce contexte (par ex : méthode d'un EJB session), sinon une nouvelle transaction est entamée et une transaction gérée par le bean est utilisée.

`org.hibernate.transaction.CMTTransactionFactory`

délègue aux transactions JTA gérées par le conteneur

Vous pouvez également définir vos propres stratégies transactionnelles (pour un service de transaction CORBA par exemple).

Certaines fonctionnalités de Hibernate (c'est-à-dire le cache de second niveau, l'association automatique des Sessions à JTA, etc.) nécessitent l'accès au `TransactionManager` JTA dans un environnement géré. Dans un serveur d'applications, vous devez indiquer comment Hibernate peut obtenir une référence vers le `TransactionManager`, car J2EE ne fournit pas un seul mécanisme standard.

Tableau 3.10. TransactionManagers JTA

Fabrique de transaction	Serveur d'applications
<code>org.hibernate.transaction.JBossTransactionManagerLookup</code>	JBoss

Fabrique de transaction	Serveur d'applications
<code>org.hibernate.transaction.WeblogicTransactionManagerLookup</code>	Weblogic
<code>org.hibernate.transaction.WebSphereTransactionManagerLookup</code>	WebSphere
<code>org.hibernate.transaction.WebSphereExtendedJTATransactionLookup</code>	WebSphere 6
<code>org.hibernate.transaction.OrionTransactionManagerLookup</code>	Orion
<code>org.hibernate.transaction.ResinTransactionManagerLookup</code>	Resin
<code>org.hibernate.transaction.JOTMTransactionManagerLookup</code>	JOTM
<code>org.hibernate.transaction.JOnASTransactionManagerLookup</code>	JOnAS
<code>org.hibernate.transaction.JRun4TransactionManagerLookup</code>	JRun4
<code>org.hibernate.transaction.BESTransactionManagerLookup</code>	Borland ES

3.8.2. `SessionFactory` associée au JNDI

Une `SessionFactory` Hibernate associée au JNDI peut simplifier l'accès à la fabrique et donc la création de nouvelles `Sessions`. Notez que cela n'est pas lié avec les `Datasource` associées au JNDI, elles utilisent juste le même registre !

Si vous désirez associer la `SessionFactory` à un nom JNDI, spécifiez un nom (par ex. `java:hibernate/SessionFactory`) en utilisant la propriété `hibernate.session_factory_name`. Si cette propriété est omise, la `SessionFactory` ne sera pas associée au JNDI (c'est particulièrement pratique dans les environnements ayant une implémentation JNDI par défaut en lecture seule, comme c'est le cas pour Tomcat).

Lorsqu'il associe la `SessionFactory` au JNDI, Hibernate utilisera les valeurs de `hibernate.jndi.url`, `hibernate.jndi.class` pour instancier un contexte d'initialisation. S'ils ne sont pas spécifiés, l'`InitialContext` par défaut sera utilisé.

Hibernate va automatiquement placer la `SessionFactory` dans JNDI après avoir appelé `cfg.buildSessionFactory()`. Cela signifie que vous devez avoir cet appel dans un code de démarrage (ou dans une classe utilitaire) dans votre application sauf si vous utilisez le déploiement JMX avec le service `HibernateService` présenté plus tard dans ce document.

Si vous utilisez `SessionFactory` JNDI, un EJB ou n'importe quelle autre classe peut obtenir la `SessionFactory` en utilisant une recherche JNDI.

Nous recommandons de lier la `SessionFactory` à JNDI dans les environnements gérés et d'utiliser un singleton `static` si ce n'est pas le cas. Pour isoler votre application de ces détails, nous vous recommandons aussi de masquer le code de recherche actuel pour une `SessionFactory` dans une classe helper, comme `HibernateUtil.getSessionFactory()`. Notez qu'une telle classe est aussi un moyen efficace de démarrer Hibernate - voir chapitre 1.

3.8.3. Gestion du contexte de la session courante à JTA

The easiest way to handle `Sessions` and transactions is Hibernate's automatic "current" `Session` management. For a discussion of contextual sessions see [Section 2.5, « Sessions contextuelles](#)

». Using the "jta" session context, if there is no Hibernate `Session` associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The `Sessions` retrieved via `getCurrentSession()` in the "jta" context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the `Sessions` to be managed by the life cycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through `UserTransaction`, or (recommended for portable code) use the Hibernate `Transaction` API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

3.8.4. Déploiement JMX

La ligne `cfg.buildSessionFactory()` doit toujours être exécutée quelque part pour obtenir une `SessionFactory` dans JNDI. Vous pouvez faire cela dans un bloc d'initialisation `static` (comme celui qui se trouve dans la classe `HibernateUtil`) ou vous pouvez déployer Hibernate en temps que *service géré*.

Hibernate est distribué avec `org.hibernate.jmx.HibernateService` pour le déploiement sur un serveur d'applications avec le support de JMX comme JBoss AS. Le déploiement et la configuration sont spécifiques à chaque vendeur. Voici un fichier `jboss-service.xml` d'exemple pour JBoss 4.0.x :

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
      name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends
>jboss.jca:service=RARDeployer</depends>
    <depends
>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName"
>java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource"
>java:HsqlDS</attribute>
    <attribute name="Dialect"
>org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled"
>true</attribute>
```

```
<attribute name="AutoCloseSessionEnabled"
>true</attribute>

<!-- Fetching options -->
<attribute name="MaximumFetchDepth"
>5</attribute>

<!-- Second-level caching -->
<attribute name="SecondLevelCacheEnabled"
>true</attribute>
<attribute name="CacheProviderClass"
>org.hibernate.cache.EhCacheProvider</attribute>
<attribute name="QueryCacheEnabled"
>true</attribute>

<!-- Logging -->
<attribute name="ShowSqlEnabled"
>true</attribute>

<!-- Mapping files -->
<attribute name="MapResources"
>auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
>
```

Ce fichier est déployé dans un répertoire `META-INF` et est empaqueté dans un fichier JAR avec l'extension `.sar` (service archive). Vous devez également empaqueter Hibernate, les librairies tierces requises, vos classes persistantes compilées et vos fichiers de mappage dans la même archive. Vos beans entreprise (souvent des EJB session) peuvent rester dans leur propre fichier JAR mais vous pouvez inclure ce fichier JAR dans le jar principal du service pour avoir une seule unité déployable à chaud. Vous pouvez consulter la documentation de JBoss AS pour plus d'informations sur les services JMX et le déploiement des EJB.

Classes persistantes

Les classes persistantes sont les classes d'une application qui implémentent les entités d'un problème métier (par ex. Client et Commande dans une application de commerce électronique). Toutes les instances d'une classe persistante ne sont pas forcément dans l'état persistant - en revanche, une instance peut être éphémère (transient) ou détachée.

Hibernate fonctionne de manière optimale lorsque ces classes suivent quelques règles simples, aussi connues comme le modèle de programmation Plain Old Java Object (POJO). Cependant, aucune de ces règles ne sont des besoins absolus. En effet, Hibernate3 présuppose très peu de choses à propos de la nature de vos objets persistants. Vous pouvez exprimer un modèle de domaine par d'autres moyens : utiliser des arbres d'instances de `Map`, par exemple.

4.1. Un exemple simple de POJO

Toute bonne application Java nécessite une classe persistante représentant les félins. Par exemple :

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

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

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

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
```

```
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
        kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

On explore quatre règles principales de classes persistantes en détail dans les sections qui suivent :

4.1.1. Implémenter un constructeur sans argument

Cat a un constructeur sans argument. Toutes les classes persistantes doivent avoir un constructeur par défaut (lequel peut ne pas être public) pour que Hibernate puisse les instancier en utilisant `Constructor.newInstance()`. Nous recommandons fortement d'avoir un constructeur par défaut avec au moins une visibilité *paquetage* pour la génération du proxy à l'exécution dans Hibernate.

4.1.2. Fournir une propriété d'identifiant (optionnel)

`Cat` possède une propriété appelée `id`. Cette propriété mappe la valeur de la colonne de clé primaire de la table d'une base de données. La propriété aurait pu s'appeler autrement, et son type aurait pu être n'importe quel type primitif, n'importe quel "encapsuleur" de type primitif, `java.lang.String` ou `java.util.Date`. (Si votre base de données héritée possède des clés composites, elles peuvent être mappées en utilisant une classe définie par l'utilisateur et possédant les propriétés associées aux types de la clé composite - voir la section concernant les identifiants composites ultérieurement).

La propriété d'identifiant est strictement optionnelle. Vous pouvez l'oublier et laisser Hibernate s'occuper des identifiants de l'objet en interne. Toutefois, ce n'est pas recommandé.

En fait, quelques fonctionnalités ne sont disponibles que pour les classes déclarant un identifiant de propriété :

- Transitive reattachment for detached objects (cascade update or cascade merge) - see [Section 10.11, « Persistance transitive »](#)
- `Session.saveOrUpdate()`
- `Session.merge()`

Nous recommandons que vous déclariez les propriétés d'identifiant de manière uniforme. Nous recommandons également que vous utilisiez un type nullable (c'est-à-dire non primitif).

4.1.3. Favoriser les classes non finales (optionnel)

Une fonctionnalité clé de Hibernate, les *proxies*, nécessitent que la classe persistante soit non finale ou qu'elle soit l'implémentation d'une interface qui déclare toutes les méthodes publiques.

Vous pouvez persister, grâce à Hibernate, les classes `final` qui n'implémentent pas d'interface, mais vous ne pourrez pas utiliser les proxies pour les chargements d'associations paresseux - ce qui limitera vos possibilités d'ajustement des performances.

Vous devriez aussi éviter de déclarer des méthodes `public final` sur des classes non-finales. Si vous voulez utiliser une classe avec une méthode `public final`, vous devez explicitement désactiver les proxies en paramétrant `lazy="false"`.

4.1.4. Déclarer les accesseurs et mutateurs des attributs persistants (optionnel)

`Cat` déclare des mutateurs pour tous ses champs persistants. Beaucoup d'autres solutions de mappage Objet/relationnel persistent directement les variables d'instance. Nous pensons qu'il est préférable de fournir une indirection entre le schéma relationnel et les structures de données internes de la classe. Par défaut, Hibernate persiste les propriétés suivant le style JavaBean, et

reconnaît les noms de méthodes de la forme `getFoo`, `isFoo` et `setFoo`. Vous pouvez changer pour un accès direct aux champs pour des propriétés particulières, si besoin est.

Les propriétés *n'ont pas* à être déclarées publiques - Hibernate peut persister une propriété avec une paire de getter/setter par défaut, `protected` ou `private`.

4.2. Implémenter l'héritage

Une sous-classe doit également suivre la première et la seconde règle. Elle hérite sa propriété d'identifiant de la classe mère `Cat`. Par exemple :

```
package eg;

public class DomesticCat extends Cat {
    private String name;

    public String getName() {
        return name;
    }

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

4.3. Implémenter `equals()` et `hashCode()`

Vous devez surcharger les méthodes `equals()` et `hashCode()` si vous :

- avez l'intention de mettre des instances de classes persistantes dans un `Set` (la manière recommandée pour représenter des associations pluri-valuées); et
- avez l'intention d'utiliser le rattachement d'instances détachées

Hibernate garantit l'équivalence de l'identité persistante (ligne de base de données) et l'identité Java seulement à l'intérieur de la portée d'une session particulière. Donc dès que nous mélangeons des instances venant de différentes sessions, nous devons implémenter `equals()` et `hashCode()` si nous souhaitons avoir une sémantique correcte pour les `Set` s.

La manière la plus évidente est d'implémenter `equals()/hashCode()` en comparant la valeur de l'identifiant des deux objets. Si cette valeur est identique, les deux doivent représenter la même ligne de base de données, ils sont donc égaux (si les deux sont ajoutés à un `Set`, nous n'aurons qu'un seul élément dans le `Set`). Malheureusement, nous ne pouvons pas utiliser cette approche avec des identifiants générés ! Hibernate n'assignera de valeur d'identifiant qu'aux objets qui sont persistants, une instance nouvellement créée n'aura donc pas de valeur d'identifiant ! De plus, si une instance est non sauvegardée et actuellement dans un `Set`, le sauvegarder assignera une valeur d'identifiant à l'objet. Si `equals()` et `hashCode()` sont basées sur la valeur de l'identifiant, le code de hachage devrait changer, rompant le contrat du `Set`. Consultez le site web de Hibernate

pour des informations plus approfondies. Notez que ceci n'est pas un problème Hibernate, mais concerne la sémantique normale de Java pour l'identité et l'égalité d'un objet.

Nous recommandons donc d'implémenter `equals()` et `hashCode()` en utilisant *l'égalité par clé métier*. L'égalité par clé métier signifie que la méthode `equals()` compare uniquement les propriétés qui forment une clé métier, une clé qui identifierait notre instance dans le monde réel (une clé candidate *naturelle*) :

```
public class Cat {

    ...

    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

A business key does not have to be as solid as a database primary key candidate (see [Section 12.1.3, « L'identité des objets »](#)). Immutable or unique properties are usually good candidates for a business key.

4.4. Modèles dynamiques



Remarque

The following features are currently considered experimental and may change in the near future.

Les entités persistantes ne doivent pas nécessairement être représentées comme des classes POJO ou des objets JavaBean à l'exécution. Hibernate supporte aussi les modèles dynamiques (en utilisant des `Map` s de `Map` s à l'exécution) et la représentation des entités comme des arbres DOM4J. Avec cette approche, vous n'écrivez pas de classes persistantes, seulement des fichiers de mappage.

By default, Hibernate works in normal POJO mode. You can set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option (see [Tableau 3.3, « Propriétés de configuration Hibernate »](#)).

Les exemples suivants démontrent la représentation utilisant des `Map` s. D'abord, dans le fichier de mappage, un `entity-name` doit être déclaré au lieu (ou en plus) d'un nom de classe :

```
<hibernate-mapping>

  <class entity-name="Customer">

    <id name="id"
        type="long"
        column="ID">
      <generator class="sequence"/>
    </id>

    <property name="name"
        column="NAME"
        type="string"/>

    <property name="address"
        column="ADDRESS"
        type="string"/>

    <many-to-one name="organization"
        column="ORGANIZATION_ID"
        class="Organization"/>

    <bag name="orders"
        inverse="true"
        lazy="false"
        cascade="all">
      <key column="CUSTOMER_ID"/>
      <one-to-many class="Order"/>
    </bag>

  </class>

</hibernate-mapping>
>
```

Notez que même si des associations sont déclarées en utilisant des noms de classe cible, le type de cible d'une association peut aussi être une entité dynamique au lieu d'un POJO.

Après avoir configuré le mode d'entité par défaut à `dynamic-map` pour la `SessionFactory`, nous pouvons lors de l'exécution fonctionner avec des `Map` s de `Map` s :

```
Session s = openSession();
Transaction tx = s.beginTransaction();

// Create a customer
Map david = new HashMap();
```



```

david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();

```

Les avantages d'un mappage dynamique sont un gain de temps pour le prototypage sans la nécessité d'implémenter les classes d'entité. Pourtant, vous perdez la vérification du typage au moment de la compilation et aurez plus d'exceptions à gérer lors de l'exécution. Grâce au mappage de Hibernate, le schéma de la base de données peut facilement être normalisé et solidifié, permettant de rajouter une implémentation propre du modèle de domaine plus tard.

Les modes de représentation d'une entité peuvent aussi être configurés en se basant sur `Session` :

```

Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession

```

Veuillez noter que l'appel à `getSession()` en utilisant un `EntityMode` se fait sur l'API `Session`, et non sur `SessionFactory`. De cette manière, la nouvelle `Session` partage les connexions JDBC, transactions et autres informations de contexte sous-jacentes. Cela signifie que vous n'avez pas à appeler `flush()` et `close()` sur la `Session` secondaire, et laissez aussi la gestion de la transaction et de la connexion à l'unité de travail primaire.

More information about the XML representation capabilities can be found in [Chapitre 19, Mappage XML](#).

4.5. Tuplizers

`org.hibernate.tuple.Tuplizer`, et ses sous-interfaces, sont responsables de la gestion d'une représentation particulière d'un fragment de données, en fonction du

`org.hibernate.EntityMode` de représentation. Si un fragment donné de données est considéré comme une structure de données, alors un tuplizer sait comment créer une telle structure de données, comment extraire des valeurs et injecter des valeurs dans une telle structure de données. Par exemple, pour le mode d'entité POJO, le tuplizer correspondant sait comment créer le POJO à travers son constructeur et comment accéder aux propriétés du POJO utilisant les accesseurs de la propriété définie.

Il y a deux types de Tuplizers de haut niveau, représentés par les interfaces `org.hibernate.tuple.EntityTuplizer` et `org.hibernate.tuple.ComponentTuplizer`. Les `EntityTuplizer` s sont responsables de la gestion des contrats mentionnés ci-dessus pour les entités, alors que les `ComponentTuplizer` s s'occupent des composants.

Les utilisateurs peuvent aussi brancher leurs propres tuplizers. Il vous faudra peut-être utiliser une implémentation de `java.util.Map` autre que `java.util.HashMap` dans le mode d'entité `dynamic-map` ; ou peut-être aurez vous besoin de définir une stratégie de génération de proxy différente de celle utilisée par défaut. Les deux devraient être effectuées en définissant une implémentation de tuplizer utilisateur. Les définitions de tuplizers sont attachées au mappage de l'entité ou du composant qu'ils devraient gérer. Revenons à l'exemple de notre entité utilisateur :

```
<hibernate-mapping>
  <class entity-name="Customer">
    <!--
      Override the dynamic-map entity-mode
      tuplizer for the customer entity
    -->
    <tuplizer entity-mode="dynamic-map"
      class="CustomMapTuplizerImpl"/>

    <id name="id" type="long" column="ID">
      <generator class="sequence"/>
    </id>

    <!-- other properties -->
    ...
  </class>
</hibernate-mapping>

public class CustomMapTuplizerImpl
  extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer {
  // override the buildInstantiator() method to plug in our custom map...
  protected final Instantiator buildInstantiator(
    org.hibernate.mapping.PersistentClass mappingInfo) {
    return new CustomMapInstantiator( mappingInfo );
  }

  private static final class CustomMapInstantiator
    extends org.hibernate.tuple.DynamicMapInstantiator {
    // override the generateMap() method to return our custom map...
    protected final Map generateMap() {
      return new CustomMap();
    }
  }
}
```

```
}
```

4.6. EntityNameResolvers

L'interface `org.hibernate.EntityNameResolver` représente un contrat pour résoudre le nom de l'entité d'une instance d'entité donnée. L'interface définit une méthode simple `resolveEntityName`, à qui l'on passe l'instance d'entité et qui doit retourner le nom d'entité qui convient (null est accepté et indiquerait que le resolver ne sait pas comment résoudre le nom de l'entité de l'instance d'entité donnée). Normalement, un `org.hibernate.EntityNameResolver` est surtout utile pour les modèles dynamiques. Vous pourriez, par exemple, utiliser des interfaces proxy comme modèle de domaine. La suite de test Hibernate comprend un exemple de ce style précis d'utilisation dans `org.hibernate.test.dynamicentity.tuplizer2`. Vous trouverez ci dessous une illustration du code de ce package.

```
/**
 * A very trivial JDK Proxy InvocationHandler implementation where we proxy an interface as
 * the domain model and simply store persistent state in an internal Map. This is an extremely
 * trivial example meant only for illustration.
 */
public final class DataProxyHandler implements InvocationHandler {
    private String entityName;
    private HashMap data = new HashMap();

    public DataProxyHandler(String entityName, Serializable id) {
        this.entityName = entityName;
        data.put( "Id", id );
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String methodName = method.getName();
        if ( methodName.startsWith( "set" ) ) {
            String propertyName = methodName.substring( 3 );
            data.put( propertyName, args[0] );
        }
        else if ( methodName.startsWith( "get" ) ) {
            String propertyName = methodName.substring( 3 );
            return data.get( propertyName );
        }
        else if ( "toString".equals( methodName ) ) {
            return entityName + "#" + data.get( "Id" );
        }
        else if ( "hashCode".equals( methodName ) ) {
            return new Integer( this.hashCode() );
        }
        return null;
    }

    public String getEntityName() {
        return entityName;
    }

    public HashMap getData() {
        return data;
    }
}
```

```
    }
}

/**
 *
 */
public class ProxyHelper {
    public static String extractEntityName(Object object) {
        // Our custom java.lang.reflect.Proxy instances actually bundle
        // their appropriate entity name, so we simply extract it from there
        // if this represents one of our proxies; otherwise, we return null
        if ( Proxy.isProxyClass( object.getClass() ) ) {
            InvocationHandler handler = Proxy.getInvocationHandler( object );
            if ( DataProxyHandler.class.isAssignableFrom( handler.getClass() ) ) {
                DataProxyHandler myHandler = ( DataProxyHandler ) handler;
                return myHandler.getEntityName();
            }
        }
        return null;
    }

    // various other utility methods ....
}

/**
 * The EntityNameResolver implementation.
 * IMPL NOTE : An EntityNameResolver really defines a strategy for how entity names should be
 * resolved. Since this particular impl can handle resolution for all of our entities we want to
 * take advantage of the fact that SessionFactoryImpl keeps these in a Set so that we only ever
 * have one instance registered. Why? Well, when it comes time to resolve an entity name,
 * Hibernate must iterate over all the registered resolvers. So keeping that number down
 * helps that process be as speedy as possible. Hence the equals and hashCode impls
 */
public class MyEntityNameResolver implements EntityNameResolver {
    public static final MyEntityNameResolver INSTANCE = new MyEntityNameResolver();

    public String resolveEntityName(Object entity) {
        return ProxyHelper.extractEntityName( entity );
    }

    public boolean equals(Object obj) {
        return getClass().equals( obj.getClass() );
    }

    public int hashCode() {
        return getClass().hashCode();
    }
}

public class MyEntityTuplizer extends PojoEntityTuplizer {
    public MyEntityTuplizer(EntityMetamodel entityMetamodel, PersistentClass mappedEntity) {
        super( entityMetamodel, mappedEntity );
    }

    public EntityNameResolver[] getEntityNameResolvers() {
        return new EntityNameResolver[] { MyEntityNameResolver.INSTANCE };
    }
}
```

```
public String determineConcreteSubclassEntityName(Object entityInstance, SessionFactoryImplementor factory) {
    String entityName = ProxyHelper.extractEntityName( entityInstance );
    if ( entityName == null ) {
        entityName = super.determineConcreteSubclassEntityName( entityInstance, factory );
    }
    return entityName;
}

...
}
```

Pour enregistrer un `org.hibernate.EntityNameResolver`, les utilisateurs doivent soit :

1. Implement a custom *Tuplizer*, implementing the `getEntityNameResolvers` method.
2. L'enregistrer dans `org.hibernate.impl.SessionFactoryImpl` (qui est la classe d'implémentation de `org.hibernate.SessionFactory`) à l'aide de la méthode `registerEntityNameResolver`.

Mappage O/R de base

5.1. Déclaration de mappage

Les mappages objet/relationnel sont généralement définis dans un document XML. Le document de mappage est conçu pour être lisible et éditable à la main. Le langage de mappage est Java-centrique, c'est-à-dire que les mappages sont construits à partir de déclarations de classes persistantes et non à partir de déclarations de tables.

Remarquez que même si beaucoup d'utilisateurs de Hibernate préfèrent écrire les fichiers de mappages XML à la main, plusieurs outils existent pour générer ce document, notamment XDoclet, Middlegen et AndroMDA.

Commençons avec un exemple de mappage :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class name="Cat"
        table="cats"
        discriminator-value="C">

        <id name="id">
            <generator class="native"/>
        </id>

        <discriminator column="subclass"
            type="character"/>

        <property name="weight"/>

        <property name="birthdate"
            type="date"
            not-null="true"
            update="false"/>

        <property name="color"
            type="eg.types.ColorUserType"
            not-null="true"
            update="false"/>

        <property name="sex"
            not-null="true"
            update="false"/>

        <property name="litterId"
            column="litterId"
            update="false"/>

    </class>

</hibernate-mapping>
```

```
<many-to-one name="mother"
  column="mother_id"
  update="false"/>

<set name="kittens"
  inverse="true"
  order-by="litter_id">
  <key column="mother_id"/>
  <one-to-many class="Cat"/>
</set>

<subclass name="DomesticCat"
  discriminator-value="D">

  <property name="name"
    type="string"/>

</subclass>

</class>

<class name="Dog">
  <!-- mapping for Dog could go here -->
</class>

</hibernate-mapping>
>
```

Étudions le contenu du document de mappage. Nous ne décrivons que les éléments et attributs du document utilisés par Hibernate à l'exécution. Le document de mappage contient aussi des attributs et éléments optionnels qui agissent sur le schéma de base de données exporté par l'outil de génération de schéma. (Par exemple l'attribut `not-null`).

5.1.1. Doctype

Tous les mappages XML devraient utiliser le doctype indiqué. En effet vous trouverez le fichier DTD à l'URL ci-dessus, dans le répertoire `hibernate-x.x.x/src/org/hibernate` ou dans `hibernate3.jar`. Hibernate va toujours chercher la DTD dans son classpath en premier lieu. Si vous constatez des recherches de la DTD sur Internet, vérifiez votre déclaration de DTD par rapport au contenu de votre classpath.

5.1.1.1. EntityResolver

Comme mentionné précédemment, Hibernate tentera en premier lieu de résoudre les DTD dans leur classpath. Il réussit à le faire en enregistrant une implémentation personnalisée de `org.xml.sax.EntityResolver` avec le `SAXReader` qu'il utilise pour lire les fichiers xml. Cet `EntityResolver` personnalisé reconnaît deux espaces de nommage `systemId` différents :

- un espace de nommage `hibernate` est reconnu dès que le résolveur rencontre un `systemId` commençant par `http://hibernate.sourceforge.net/`. Le résolveur tente alors de résoudre ces entités via le chargeur de classe qui a chargé les classes Hibernate.

- un espace de nommage utilisateur est reconnu dès que le résolveur rencontre un `systemId` qui utilise un protocole URL `classpath://`. Le résolveur tentera alors de résoudre ces entités via (1) le chargeur de classe du contexte du thread courant et (2) le chargeur de classe qui a chargé les classes Hibernate.

Un exemple d'utilisation de l'espace de nommage utilisateur:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" 'http://
hibernate.sourceforge.net/hibernate-mapping-3.0.dtd' [
<!ENTITY version "3.5.6-Final">
<!ENTITY today "September 15, 2010">

    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">

]

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    </class>
    &types;
</hibernate-mapping>
```

Where `types.xml` is a resource in the `your.domain` package and contains a custom [typedef](#).

5.1.2. Hibernate-mappage

Cet élément a plusieurs attributs optionnels. Les attributs `schema` et `catalog` indiquent que les tables mentionnées dans ce mappage appartiennent au schéma nommé et/ou au catalogue. S'ils sont spécifiés, les noms de tables seront qualifiés par les noms de schéma et de catalogue. L'attribut `default-cascade` indique quel type de cascade sera utilisé par défaut pour les propriétés et collections qui ne précisent pas l'attribut `cascade`. L'attribut `auto-import` nous permet d'utiliser par défaut des noms de classes non qualifiés dans le langage de requête, par défaut.

```
<hibernate-mapping
    schema="schemaName"
    catalog="catalogName"
    default-cascade="cascade_style"
    default-access="field|property|ClassName"
    default-lazy="true|false"
    auto-import="true|false"
    package="package.name"
```

```
/>
```

- ❶ `schema` (optionnel) : le nom d'un schéma de base de données.
- ❷ `catalog` (optionnel) : le nom d'un catalogue de base de données.
- ❸ `default-cascade` (optionnel - par défaut vaut : `none`) : un type de cascade par défaut.
- ❹ `default-access` (optionnel - par défaut vaut : `property`) : Comment hibernate accèdera aux propriétés. On peut aussi redéfinir sa propre implémentation de `PropertyAccessor`.
- ❺ `default-lazy` (optionnel - par défaut vaut : `true`) : Valeur par défaut pour des attributs `lazy` non spécifiés des mappages de classe et de collection.
- ❻ `auto-import` (optionnel - par défaut vaut : `true`) : spécifie si l'on peut utiliser des noms de classes non qualifiés (de classes de ce mappage) dans le langage de requête.
- ❼ `package` (optionnel) : préfixe de paquetage par défaut pour les noms de classe non qualifiés du document de mappage.

Si deux classes persistantes possèdent le même nom de classe (non qualifié), vous devez configurer `auto-import="false"`. Hibernate lancera une exception si vous essayez d'assigner le même nom "importé" à deux classes.

Notez que l'élément `hibernate-mapping` vous permet d'imbriquer plusieurs mappages de `<class>` persistantes, comme dans l'exemple ci-dessus. Cependant il est recommandé (et c'est parfois une exigence de certains outils) de mapper une seule classe persistante (ou une seule hiérarchie de classes) par fichier de mappage et de nommer ce fichier d'après le nom de la superclasse persistante, par exemple `Cat.hbm.xml`, `Dog.hbm.xml`, ou en cas d'héritage, `Animal.hbm.xml`.

5.1.3. Classe

Déclarez une classe persistante avec l'élément `class`. Part exemple :

```
<class
    name="ClassName"
    table="tableName"
    discriminator-value="discriminator_value"
    mutable="true|false"
    schema="owner"
    catalog="catalog"
    proxy="ProxyInterface"
    dynamic-update="true|false"
    dynamic-insert="true|false"
    select-before-update="true|false"
    polymorphism="implicit|explicit"
    where="arbitrary sql where condition"
    persister="PersisterClass"
```

```

batch-size="N"
optimistic-lock="none|version|dirty|all"
lazy="true|false"
entity-name="EntityName"
check="arbitrary sql check condition"
rowid="rowid"
subselect="SQL expression"
abstract="true|false"
node="element-name"
/>

```

- ❶ name (optionnel) : le nom Java complet de la classe (ou interface) persistante. Si cet attribut est absent, nous supposons que ce mappage ne se rapporte pas à une entité POJO.
- ❷ table (optionnel - par défaut le nom non-qualifié de la classe) : le nom de sa table en base de données.
- ❸ discriminator-value (optionnel - par défaut le nom de la classe) : une valeur permettant de distinguer les différentes sous-classes utilisées dans le comportement polymorphe. Les valeurs `null` et `not null` sont autorisées.
- ❹ mutable (optionnel, vaut `true` par défaut) : spécifie que des instances de la classe sont (ou non) immuables.
- ❺ schema (optionnel) : surcharge le nom de schéma spécifié par l'élément racine `<hibernate-mapping>`.
- ❻ catalog (optionnel) : surcharge le nom du catalogue spécifié par l'élément racine `<hibernate-mapping>`.
- ❼ proxy (optionnel) : spécifie une interface à utiliser pour l'initialisation différée (lazy loading) des proxies. Vous pouvez indiquer le nom de la classe elle-même.
- ❽ dynamic-update (optionnel, par défaut à `false`) : spécifie que les SQL `UPDATE` doivent être générés à l'exécution et contenir uniquement les colonnes dont les valeurs ont été modifiées.
- ❾ dynamic-insert (optionnel, par défaut à `false`) : spécifie que les SQL `INSERT` doivent être générés à l'exécution et ne contenir que les colonnes dont les valeurs sont non nulles.
- ❿ select-before-update (optionnel, par défaut à `false`) : spécifie que Hibernate ne doit *jamais* exécuter un SQL `UPDATE` sans être certain qu'un objet a été réellement modifié. Dans certains cas, (en réalité, seulement quand un objet transient a été associé à une nouvelle session par `update()`), cela signifie que Hibernate exécutera un SQL `SELECT` pour déterminer si un SQL `UPDATE` est véritablement nécessaire.
- ⓫ polymorphism (optionnel, vaut `implicit` par défaut) : détermine si, pour cette classe, une requête polymorphe implicite ou explicite est utilisée.
- ⓬ where (optionnel) spécifie une clause SQL `WHERE` à utiliser lorsque l'on récupère des objets de cette classe.
- ⓭ persister (optionnel) : spécifie un `ClassPersister` particulier.
- ⓮ batch-size (optionnel, par défaut = 1) : spécifie une "taille de lot" pour remplir les instances de cette classe par identifiant en une seule requête.
- ⓯ optimistic-lock (optionnel, par défaut = `version`) : détermine la stratégie de verrouillage optimiste.

- 16 `lazy` (optionnel) : l'extraction différée (lazy fetching) peut être totalement désactivée en configurant `lazy="false"`.
- 17 `entity-name` (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See [Section 4.4, « Modèles dynamiques »](#) and [Chapitre 19, Mappage XML](#) for more information.
- 18 `check` (optionnel) : expression SQL utilisée pour générer une contrainte de vérification *check* multi-lignes pour la génération automatique de schéma.
- 19 `rowid` (optionnel) : Hibernate peut utiliser des ROWID sur les bases de données qui utilisent ce mécanisme. Par exemple avec Oracle, Hibernate peut utiliser la colonne additionnelle `rowid` pour des mise à jour rapides si cette option vaut `rowid`. Un ROWID est un détail d'implémentation et représente la localisation physique d'un uplet enregistré.
- 20 `subselect` (optionnel) : permet de mapper une entité immuable en lecture-seule sur un sous-select de base de données. Utile pour avoir une vue au lieu d'une table de base, mais à éviter. Voir plus bas pour plus d'informations.
- 21 `abstract` (optionnel) : utilisé pour marquer des superclasses abstraites dans des hiérarchies de `<union-subclass>`.

Il est tout à fait possible d'utiliser une interface comme nom de classe persistante. Vous devez alors déclarer les classes implémentant cette interface en utilisant l'élément `<subclass>`. Vous pouvez faire persister toute classe interne *static*. Vous devez alors spécifier le nom de la classe par la notation habituelle des classes internes, c'est à dire `eg. Foo$Bar`.

Les classes immuables, `mutable="false"`, ne peuvent pas être modifiées ou supprimées par l'application. Cela permet à Hibernate de faire quelques optimisations mineures sur les performances.

L'attribut optionnel `proxy` permet les initialisations différées des instances persistantes de la classe. Hibernate retournera initialement des proxies CGLIB qui implémentent l'interface nommée. Le véritable objet persistant ne sera chargé que lorsqu'une méthode du proxy sera appelée. Voir plus bas le paragraphe abordant les Proxies et leur initialisation différée (lazy initialization).

Le polymorphisme *implicite* signifie que des instances de la classe seront retournées par une requête qui utilise les noms de la classe ou de chacune de ses superclasses ou encore des interfaces implémentées par cette classe ou ses superclasses. Les instances des classes filles seront retournées par une requête qui utilise le nom de la classe elle même. Le polymorphisme *explicite* signifie que les instances de la classe ne seront retournées que par une requête qui utilise explicitement son nom et que seules les instances des classes filles déclarées dans les éléments `<subclass>` ou `<joined-subclass>` seront retournées. Dans la majorité des cas la valeur par défaut, `polymorphism="implicit"`, est appropriée. Le polymorphisme explicite est utile lorsque deux classes différentes sont mappées à la même table (ceci permet d'écrire une classe "légère" qui ne contient qu'une partie des colonnes de la table - voir la partie design pattern du site communautaire).

L'attribut `persister` vous permet de personnaliser la stratégie de persistance utilisée pour la classe. Vous pouvez, par exemple, spécifier votre propre sous-classe de `org.hibernate.persister.EntityPersister` ou vous pourriez aussi fournir une nouvelle implémentation de l'interface `org.hibernate.persister.ClassPersister` qui proposerait une persistance via, par exemple, des appels de procédures stockées, de la sérialisation vers des fichiers plats ou un annuaire LDAP. Voir `org.hibernate.test.CustomPersister` pour un exemple simple (d'une "persistance" vers une `Hashtable`).

Notez que les paramètres `dynamic-update` et `dynamic-insert` ne sont pas hérités par les sous-classes et peuvent donc être spécifiés pour les éléments `<subclass>` ou `<joined-subclass>`. Ces paramètres peuvent améliorer les performances dans certains cas, mais peuvent aussi les amoindrir. À utiliser en connaissance de causes.

L'utilisation de `select-before-update` va généralement faire baisser les performances. Ce paramètre est pratique pour éviter l'appel inutile par un déclenchement de mise à jour de base de donnée, quand on ré-attache un graphe d'instances à une `Session`.

Si vous utilisez le `dynamic-update`, les différentes stratégies de verrouillage optimiste sont les suivantes :

- `version` vérifie les colonnes version/timestamp
- `all` vérifie toutes les colonnes
- `dirty` vérifie les colonnes modifiées, permettant quelques mise à jour concurrentes
- `none` n'utilisez pas le verrouillage optimiste

Nous encourageons *très* fortement l'utilisation de colonnes de version/timestamp pour le verrouillage optimiste avec Hibernate. C'est la meilleure stratégie en ce qui concerne les performances et la seule qui gère correctement les modifications sur les instances détachées (c'est à dire lorsqu'on utilise `Session.merge()`).

Il n'y a pas de différence entre table et vue pour le mappage Hibernate, comme on peut s'y attendre, cela est transparent au niveau base de données (remarquez que certaines BDD ne supportent pas les vues correctement, notamment pour les mise à jour). Il est possible que vous souhaitiez utiliser une vue mais vous ne puissiez pas en créer une sur votre BDD (c'est-à-dire avec un schéma ancien). Dans ces cas, vous pouvez mapper une entité immuable en lecture seule sur une expression sous-select SQL donnée :

```
<class name="Summary">
  <subselect>
    select item.name, max(bid.amount), count(*)
    from item
    join bid on bid.item_id = item.id
    group by item.name
  </subselect>
```

```
<synchronize table="item"/>
<synchronize table="bid"/>
<id name="name"/>
...
</class>
>
```

Déclarez les tables à synchroniser avec cette entité pour assurer que le flush automatique se produise correctement, et pour que les requêtes sur l'entité dérivée ne renvoient pas des données périmées. Le `<subselect>` est disponible comme attribut ou comme élément de mappage imbriqué.

5.1.4. id

Les classes mappées *doivent* déclarer la clé primaire de la table en base de données. La plupart des classes auront aussi une propriété de type JavaBeans présentant l'identifiant unique d'une instance. L'élément `<id>` sert à définir le mappage entre cette propriété et la colonne de la clé primaire.

```
<id
    name="propertyName"
    type="typename"
    column="column_name"
    unsaved-value="null|any|none|undefined|id_value"
    access="field|property|ClassName">
    node="element-name|@attribute-name|element/@attribute|."
    <generator class="generatorClass"/>
</id>
>
```

- ❶ `name` (optionnel) : nom de la propriété de l'identifiant.
- ❷ `type` (optionnel) : nom indiquant le type Hibernate.
- ❸ `column` (optionnel - le nom de la propriété est pris par défaut) : nom de la colonne de la clé primaire.
- ❹ `unsaved-value` (optionnel - devient par défaut une valeur "sensible") : une valeur de propriété d'identifiant qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances détachées qui ont été sauvegardées ou chargées dans une session précédente.
- ❺ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.

Si l'attribut `name` est absent, Hibernate considère que la classe ne possède pas de propriété d'identifiant.

L'attribut `unsaved-value` n'est presque jamais nécessaire dans Hibernate3.

La déclaration alternative `<composite-id>` permet l'accès aux données d'anciens systèmes qui utilisent des clés composées. Son utilisation est fortement déconseillée pour d'autres cas.

5.1.4.1. Generator

L'élément enfant `<generator>` nomme une classe Java utilisée pour générer les identifiants uniques pour les instances des classes persistantes. Si des paramètres sont requis pour configurer ou initialiser l'instance du générateur, ils sont passés en utilisant l'élément `<param>`.

```
<id name="id" type="long" column="cat_id">
  <generator class="org.hibernate.id.TableHiLoGenerator">
    <param name="table">
      <uid_table/>
    <param name="column">
      <next_hi_value_column/>
    </generator>
  </id>
</>
```

Tous les générateurs implémentent l'interface `org.hibernate.id.IdentifiantGenerator`. C'est une interface très simple ; certaines applications peuvent proposer leurs propres implémentations spécialisées. Cependant, Hibernate propose une série d'implémentations intégrées. Il existe des noms raccourcis pour les générateurs intégrés :

`increment`

génère des identifiants de type `long`, `short` ou `int` qui ne sont uniques que si aucun autre processus n'insère de données dans la même table. *Ne pas utiliser en environnement clusterisé.*

`identity`

prend en charge les colonnes d'identité dans DB2, MySQL, MS SQL Server, Sybase et HypersonicSQL. L'identifiant renvoyé est de type `long`, `short` ou `int`.

`sequence`

utilise une séquence dans DB2, PostgreSQL, Oracle, SAP DB, McKoi ou un générateur dans Interbase. L'identifiant renvoyé est de type `long`, `short` ou `int`

`hilo`

utilise un algorithme hi/lo pour générer de façon efficace des identifiants de type `long`, `short` ou `int`, en prenant comme source de valeurs "hi" une table et une colonne (par défaut `hibernate_unique_key` et `next_hi` respectivement). L'algorithme hi/lo génère des identifiants uniques pour une base de données particulière seulement.

`seqhilo`

utilise un algorithme hi/lo pour générer efficacement des identifiants de type `long`, `short` ou `int`, en prenant une séquence en base nommée.

`uuid`

utilise un algorithme de type UUID 128 bits pour générer des identifiants de type string, unique au sein d'un réseau (l'adresse IP est utilisée). Le UUID est encodé en une chaîne de nombre hexadécimaux de longueur 32.

`guid`

utilise une chaîne GUID générée par la base pour MS SQL Server et MySQL.

`native`

choisit `identity`, `sequence` ou `hilo` selon les possibilités offertes par la base de données sous-jacente.

`assigned`

permet à l'application d'affecter un identifiant à l'objet avant que la méthode `save()` soit appelée. Il s'agit de la stratégie par défaut si aucun `<generator>` n'est spécifié.

`select`

récupère une clé primaire assignée par un déclencheur (trigger) de base de données en sélectionnant la ligne par une clé unique quelconque et en extrayant la valeur de la clé primaire.

`foreign`

utilise l'identifiant d'un autre objet associé. Habituellement utilisé en conjonction avec une association `<one-to-one>` sur la clé primaire.

`sequence-identity`

Une stratégie de génération de séquence spécialisée qui utilise une séquence de base de données pour la génération réelle de valeurs, tout en utilisant JDBC3 `getGeneratedKeys` pour retourner effectivement la valeur d'identifiant générée, comme faisant partie de l'exécution de la déclaration `insert`. Cette stratégie est uniquement prise en charge par les pilotes Oracle 10g pour JDK 1.4. Notez que les commentaires sur ces déclarations `insert` sont désactivés à cause d'un bogue dans les pilotes d'Oracle.

5.1.4.2. Algorithme Hi/lo

Les générateurs `hilo` et `seqhilo` proposent deux implémentations alternatives de l'algorithme hi/lo. La première implémentation nécessite une table "spéciale" en base pour héberger la prochaine valeur "hi" disponible. La seconde utilise une séquence de type Oracle (quand la base sous-jacente le propose).

```
<id name="id" type="long" column="cat_id">
  <generator class="hilo">
    <param name="table"
  >hi_value</param>
    <param name="column"
  >next_value</param>
    <param name="max_lo"
  >100</param>
```



```

    </generator>
</id>
>

```

```

<id name="id" type="long" column="cat_id">
  <generator class="seqhilo">
    <param name="sequence"
>hi_value</param>
    <param name="max_lo"
>100</param>
  </generator>
</id>
>

```

Malheureusement, vous ne pouvez pas utiliser `hilo` quand vous apportez votre propre Connection à Hibernate. Quand Hibernate utilise une datasource du serveur d'application pour obtenir des connexions inscrites avec JTA, vous devez correctement configurer `hibernate.transaction.manager_lookup_class`.

5.1.4.3. Algorithme UUID

Le contenu du UUID est : l'adresse IP, la date de démarrage de la JVM (précis au quart de seconde), l'heure système et une contre-valeur (unique au sein de la JVM). Il n'est pas possible d'obtenir une adresse MAC ou une adresse mémoire à partir de Java, c'est donc le mieux que l'on puisse faire sans utiliser JNI.

5.1.4.4. Colonnes identifiantes et séquences

Pour les bases qui implémentent les colonnes "identité" (DB2, MySQL, Sybase, MS SQL), vous pouvez utiliser la génération de clé par `identity`. Pour les bases qui implémentent les séquences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) vous pouvez utiliser la génération de clé par `sequence`. Ces deux méthodes nécessitent deux requêtes SQL pour insérer un nouvel objet. Par exemple :

```

<id name="id" type="long" column="person_id">
  <generator class="sequence">
    <param name="sequence"
>person_id_sequence</param>
  </generator>
</id>
>

```

```

<id name="id" type="long" column="person_id" unsaved-value="0">
  <generator class="identity"/>
</id>
>

```

Pour le développement multi-plateformes, la stratégie `native` choisira entre les méthodes `identity`, `sequence` et `hilo`, selon les possibilités offertes par la base sous-jacente.

5.1.4.5. Identifiants assignés

Si vous souhaitez que l'application assigne des identifiants (par opposition à la génération par Hibernate), vous pouvez utiliser le générateur `assigned`. Ce générateur spécial utilisera une valeur d'identifiant déjà utilisée par la propriété identifiant l'objet. Ce générateur est utilisé quand la clé primaire est une clé naturelle plutôt qu'une clé secondaire. C'est le comportement par défaut si vous ne précisez pas d'élément `<generator>`.

Choisir le générateur `assigned` fait que Hibernate utilise `unsaved-value="undefined"`, le forçant ainsi à interroger la base de données pour déterminer si une instance est transiente ou détachée, à moins d'utiliser une propriété `version` ou `timestamp`, ou alors de définir `Interceptor.isUnsaved()`.

5.1.4.6. Clés primaires assignées par les triggers

Pour les schémas de base hérités d'anciens systèmes uniquement (Hibernate ne génère pas de DDL avec des triggers)

```
<id name="id" type="long" column="person_id">
  <generator class="select">
    <param name="key">
      >socialSecurityNumber</param>
    </generator>
  </id>
</>
```

Dans l'exemple ci-dessus, il y a une valeur de propriété unique appelée `socialSecurityNumber`. Elle est définie par la classe en tant que clé naturelle et il y a également une clé secondaire appelée `person_id` dont la valeur est générée par un trigger.

5.1.5. La méthode `getter` de l'identifiant

A partir de la version 3.2.3, 2 générateurs représentent une nouvelle conception de 2 aspects séparés de la génération d'identifiants. Le premier aspect est la portabilité de la base de données; le second est l'optimization, c'est à dire que vous n'avez pas à interroger la base de données pour chaque requête de valeur d'identifiant. Ces deux nouveaux générateurs sont sensés prendre la place de générateurs décrits ci-dessus, ayant pour préfixe 3.3.x. Cependant, ils sont inclus dans les versions actuelles, et peuvent être référencés par FQN.

Le premier de ces nouveaux générateurs est `org.hibernate.ID.Enhanced.SequenceStyleGenerator` qui est destiné, tout d'abord, comme un remplacement pour le générateur `séquence` et, deuxièmement, comme un générateur de portabilité supérieur à `natif`. C'est parce que `natif` a généralement le choix entre `identité` et `séquence` qui ont des sémantiques largement différentes, ce qui peut entraîner

des problèmes subtils en observant la portabilité des applications. `org.hibernate.ID.Enhanced SequenceStyleGenerator`., cependant, réalise la portabilité d'une manière différente. Il choisit entre une table ou une séquence dans la base de données pour stocker ses valeurs s'incrémentant, selon les capacités du dialecte utilisé. La différence avec `natif` c'est que de stockage basé sur les tables ou basé sur la séquence ont la même sémantique. En fait, les séquences sont exactement ce qu'Hibernate essaie d'émuler avec ses générateurs basée sur les tables. Ce générateur a un certain nombre de paramètres de configuration :

- `sequence_name` (en option, par défaut = `hibernate_sequence`): le nom de la séquence ou table à utiliser.
- `initial_value` (en option - par défaut = 1) : la première valeur à extraire de la séquence/table. En termes de création de séquences, c'est semblable à la clause qui s'appelle "STARTS WITH" normalement.
- `increment_size` (en option - par défaut = 1): la valeur par laquelle les appels suivants à la séquence / table doivent différer. En termes de création de séquence, c'est analogue à la clause généralement nommé "INCREMENT BY".
- `force_table_use` (optionnel - par défaut = `false`) : doit-on forcer l'utilisation de la table en tant que structure de soutien même si le dialecte peut supporter la séquence ?
- `value_column` (en option - par défaut = `next_val`): uniquement utile pour les structures de tables. Correspond au nom de la colonne de la table qui est utilisée pour contenir la valeur.
- `optimizer` (optional - defaults to `none`): See [Section 5.1.6, « Optimisation du générateur d'identifiants »](#)

Le deuxième de ces nouveaux générateurs est `org.hibernate.ID.Enhanced.TableGenerator`, qui est destiné, tout d'abord, comme un remplacement pour le générateur de la table, même si elle fonctionne effectivement beaucoup plus comme `org.hibernate.ID.MultipleHiLoPerTableGenerator` et deuxièmement, comme une remise en œuvre de `org.hibernate.ID.MultipleHiLoPerTableGenerator`, qui utilise la notion d'optimizers enfichables. Essentiellement ce générateur définit une table susceptible de contenir un certain nombre de valeurs d'incrément différents simultanément à l'aide de plusieurs lignes distinctement masquées. Ce générateur a un certain nombre de paramètres de configuration :

- `table_name` (en option - valeur par défaut = `hibernate_sequences`): le nom de la table à utiliser.
- `value_column_name` (en option - valeur par défaut = `next_val`): le nom de la colonne contenue dans la table utilisée pour la valeur.
- `segment_column_name` (en option - par défaut = `sequence_name`): le nom de la colonne de la table qui est utilisée pour contenir la "segment key". Il s'agit de la valeur qui identifie la valeur d'incrément à utiliser.
- `segment_value` (en option - par défaut = `par défaut`): La "segment key" valeur pour le segment à partir de laquelle nous voulons extraire des valeurs d'incrément pour ce générateur.
- `segment_value_length` (en option - par défaut = 255): Utilisée pour la génération de schéma ; la taille de la colonne pour créer cette colonne de clé de segment.
- `initial_value` (en option - par défaut est 1 : La valeur initiale à récupérer à partir de la table.
- `increment_size` (en option - par défaut = 1): La valeur par laquelle les appels à la table, qui suivent, devront différer.

- `optimizer` (optional - defaults to `''`): See [Section 5.1.6, « Optimisation du générateur d'identifiants »](#)

5.1.6. Optimisation du générateur d'identifiants

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value group. This is the role of the pluggable optimizers. Currently only the two enhanced generators ([Section 5.1.5, « La méthode `getter` de l'identifiant »](#)) support this operation.

- `aucun` (en général il s'agit de la valeur par défaut si aucun optimizer n'a été spécifié): n'effectuera pas d'optimisations et n'interrogera pas la base de données à chaque demande.
- `hilo`: applique un algorithme hi/lo autour des valeurs extraites des base de données. Les valeurs de la base de données de cet optimizer sont censées être séquentielles. Les valeurs extraites de la structure des base de données pour cet optimizer indique le "numéro de groupe". Le `increment_size` est multiplié par cette valeur en mémoire pour définir un groupe de "hi value".
- `mise en commun`: tout comme dans le cas de `hilo`, cet optimizer tente de réduire le nombre d'interrogations vers la base de données. Ici, cependant, nous avons simplement stocké la valeur de départ pour le "prochain groupe" dans la structure de la base de données plutôt qu'une valeur séquentielle en combinaison avec un algorithme de regroupement en mémoire. Ici, `increment_size` fait référence aux valeurs provenant de la base de données.

5.1.7. composite-id

```
<composite-id
  name="propertyName"
  class="ClassName"
  mapped="true|false"
  access="field|property|ClassName">
  node="element-name|. "

  <key-property name="propertyName" type="typename" column="column_name"/>
  <key-many-to-one name="propertyName" class="ClassName" column="column_name"/>
  .....
</composite-id>
>
```

Pour une table avec clé composée, vous pouvez mapper plusieurs attributs de la classe comme propriétés identifiantes. L'élément `<composite-id>` accepte les mappages de propriétés `<key-property>` et les mappages `<key-many-to-one>` comme éléments enfants.

```
<composite-id>
  <key-property name="medicareNumber"/>
  <key-property name="dependent"/>
</composite-id>
```

```
</composite-id>
>
```

Vos classes persistantes *doivent* surcharger les méthodes `equals()` et `hashCode()` pour implémenter l'égalité d'identifiant composite. Elles doivent aussi implémenter l'interface `Serializable`.

Malheureusement, cette approche signifie qu'un objet persistant est son propre identifiant. Il n'y a pas d'autre moyen pratique de "manipuler" l'objet que par l'objet lui-même. Vous devez instancier une instance de la classe persistante elle-même et peupler ses attributs identifiants avant de pouvoir appeler la méthode `load()` pour charger son état persistant associé à une clé composée. Nous appelons cette approche "identifiant composé *embarqué*" et ne la recommandons pas pour des applications complexes.

Une seconde approche, appelée identifiant composé *mappé*, consiste à dupliquer les propriétés identifiantes nommées dans l'élément `<composite-id>` à la fois dans la classe persistante et dans une classe identifiante particulière.

```
<composite-id class="MedicareId" mapped="true">
  <key-property name="medicareNumber" />
  <key-property name="dependent" />
</composite-id>
>
```

Dans cet exemple, la classe d'identifiant composée, `MedicareId` et la classe mappée elle-même, possèdent les propriétés `medicareNumber` et `dependent`. La classe identifiante doit redéfinir `equals()` et `hashCode()` et implémenter `Serializable`. Le désavantage de cette approche est la duplication du code.

Les attributs suivants servent à configurer un identifiant composé mappé :

- `mapped` (optionnel, défaut à `false`) : indique qu'un identifiant composé mappé est utilisé, et que les mappages de propriétés contenues font référence aux deux classes, la classe mappée et la classe identifiante composée.
- `class` (optionnel, mais requis pour un identifiant composé mappé) : la classe utilisée comme identifiant composé.

We will describe a third, even more convenient approach, where the composite identifier is implemented as a component class in [Section 8.4, « Les composants en tant qu'identifiants composés »](#). The attributes described below apply only to this alternative approach:

- `name` (optionnel, requis pour cette approche) : une propriété de type composant qui contient l'identifiant composé (voir chapitre 9).
- `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.

- `class` (optionnel - par défaut le type de la propriété déterminé par réflexion) : la classe composant utilisée comme identifiant (voir prochaine section).

La troisième approche, un *composant d'identifiant*, est celle que nous recommandons pour toutes vos applications.

5.1.8. Discriminator

L'élément `<discriminator>` est nécessaire pour la persistance polymorphique qui utilise la stratégie de mappage de table par hiérarchie de classe et déclare une colonne discriminante de la table. La colonne discriminante contient des valeurs marqueur qui permettent à la couche de persistance de savoir quelle sous-classe instancier pour une ligne particulière de table en base. Un nombre restreint de types peuvent être utilisés : `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
    column="discriminator_column"
    type="discriminator_type"
    force="true|false"
    insert="true|false"
    formula="arbitrary sql expression"
/>
```

①
②
③
④
⑤

- ① `column` (optionnel - par défaut à `class`), le nom de la colonne discriminante.
- ② `type` (optionnel - par défaut à `string`) un nom indiquant le type Hibernate.
- ③ `force` (optionnel - par défaut à `false`) "oblige" Hibernate à spécifier une valeur discriminante autorisée même quand on récupère toutes les instances de la classe de base.
- ④ `insert` (optionnel - par défaut à `true`) à passer à `false` si la colonne discriminante fait aussi partie d'un identifiant composé mappé (Indique à Hibernate de ne pas inclure la colonne dans les SQL `INSERT` s).
- ⑤ `formula` (optionnel) une expression SQL arbitraire qui est exécutée quand un type doit être évalué. Permet la discrimination basée sur le contenu.

Les véritables valeurs de la colonne discriminante sont spécifiées par l'attribut `discriminator-value` des éléments `<class>` et `<subclass>`.

L'attribut `force` n'est utile que si la table contient des lignes avec des valeurs "extra" discriminantes qui ne sont pas mappées à une classe persistante. Ce ne sera généralement pas le cas.

En utilisant l'attribut `formula` vous pouvez déclarer une expression SQL arbitraire qui sera utilisée pour évaluer le type d'une ligne :

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
```

```
type="integer"/>
```

5.1.9. Version (optionnel)

L'élément `<version>` est optionnel et indique que la table contient des données versionnées. C'est particulièrement utile si vous avez l'intention d'utiliser des *transactions longues* (voir plus-bas).

```
<version
    column="version_column"
    name="propertyName"
    type="typename"
    access="field|property|ClassName"
    unsaved-value="null|negative|undefined"
    generated="never|always"
    insert="true|false"
    node="element-name|@attribute-name|element/@attribute|."
/>
```

- ❶ `column` (optionnel - par défaut égal au nom de la propriété) : le nom de la colonne contenant le numéro de version.
- ❷ `name` : le nom d'un attribut de la classe persistante.
- ❸ `type` (optionnel - par défaut à `integer`) : le type du numéro de version.
- ❹ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ❺ `unsaved-value` (optionnel - par défaut à `undefined`) : une valeur de la propriété d'identifiant qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances détachées qui ont été sauvegardées ou chargées dans une session précédente. `Undefined` indique que la valeur de la propriété identifiant devrait être utilisée.
- ❻ `generated` (optional - defaults to `never`): specifies that this version property value is generated by the database. See the discussion of [generated properties](#) for more information.
- ❼ `insert` (optionnel - par défaut à `true`) : indique si la colonne de version doit être incluse dans les ordres SQL insert. Peut être configuré à `false` si et seulement si la colonne de la base de données est définie avec une valeur par défaut égale à 0.

Les numéros de version doivent avoir les types Hibernate `long`, `integer`, `short`, `timestamp` ou `calendar`.

Une propriété de version ou un timestamp ne doit jamais être null pour une instance détachée, ainsi Hibernate pourra détecter toute instance ayant une version ou un timestamp null comme transient, quelles que soient les stratégies `unsaved-value` spécifiées. *Déclarer un numéro de version ou un timestamp "nullable" est un moyen pratique d'éviter tout problème avec les ré-attachements transitifs dans Hibernate, particulièrement utile pour ceux qui utilisent des identifiants assignés ou des clés composées.*

5.1.10. Timestamp (optionnel)

L'élément optionnel `<timestamp>` indique que la table contient des données horodatées (timestamped). Cela sert d'alternative à l'utilisation de numéros de version. Les timestamps (ou horodatage) sont par nature une implémentation moins fiable pour le verrouillage optimiste. Cependant, l'application peut parfois utiliser l'horodatage à d'autres fins.

```
<timestamp
    column="timestamp_column"
    name="propertyName"
    access="field|property|ClassName"
    unsaved-value="null|undefined"
    source="vm|db"
    generated="never|always"
    node="element-name|@attribute-name|element/@attribute|."
/>
```

- ❶ `column` (optionnel - par défaut devient le nom de la propriété) : le nom d'une colonne contenant le timestamp.
- ❷ `name` : le nom d'une propriété au sens JavaBean de type `Java Date` ou `Timestamp` de la classe persistante.
- ❸ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ❹ `unsaved-value` (optionnel - par défaut à `null`) : propriété dont la valeur est un numéro de version qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances détachées qui ont été sauvegardées ou chargées dans une session précédente. (`undefined` indique que la valeur de propriété identifiant devrait être utilisée).
- ❺ `source` (optionnel - par défaut à `vm`) : d'où Hibernate doit-il récupérer la valeur du timestamp? Depuis la base de données ou depuis la JVM d'exécution? Les valeurs de timestamp de la base de données provoquent une surcharge puisque Hibernate doit interroger la base pour déterminer la prochaine valeur mais cela est plus sûr lorsque vous fonctionnez dans un cluster. Remarquez aussi que certains des `Dialect` s ne supportent pas cette fonction, et que d'autres l'implémentent mal, à cause d'un manque de précision (Oracle 8 par exemple).
- ❻ `generated` (optional - defaults to `never`): specifies that this timestamp property value is actually generated by the database. See the discussion of [generated properties](#) for more information.



Note

Notez que `<timestamp>` est équivalent à `<version type="timestamp">` et `<timestamp source="db">` équivaut à `<version type="dbtimestamp">`

5.1.11. Property

L'élément `<property>` déclare une propriété persistante de la classe au sens JavaBean.

```
<property
    name="propertyName"
    column="column_name"
    type="typename"
    update="true|false"
    insert="true|false"
    formula="arbitrary SQL expression"
    access="field|property|ClassName"
    lazy="true|false"
    unique="true|false"
    not-null="true|false"
    optimistic-lock="true|false"
    generated="never|insert|always"
    node="element-name|@attribute-name|element/@attribute|. "
    index="index_name"
    unique_key="unique_key_id"
    length="L"
    precision="P"
    scale="S"
/>
```

- ❶ `name` : nom de la propriété, avec une lettre initiale en minuscule.
- ❷ `column` (optionnel - par défaut au nom de la propriété) : le nom de la colonne mappée. Cela peut aussi être indiqué dans le(s) sous-élément(s) `<column>` imbriqués.
- ❸ `type` (optionnel) : nom indiquant le type Hibernate.
- ❹ `update`, `insert` (optionnel - par défaut à `true`) : indique que les colonnes mappées devraient être incluses dans des déclarations SQL `UPDATE` et/ou des `INSERT`. Mettre les deux à `false` autorise une propriété pure dérivée dont la valeur est initialisée de quelque autre propriété qui mappe à la même colonne(s) ou par un trigger ou une autre application. (utile si vous savez qu'un trigger affectera la valeur à la colonne).
- ❺ `formula` (optionnel) : une expression SQL qui définit la valeur pour une propriété *calculée*. Les propriétés calculées ne possèdent pas leur propre mappage.
- ❻ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ❼ `lazy` (optionnel - par défaut à `false`) : indique que cette propriété devrait être chargée en différé (lazy loading) quand on accède à la variable d'instance pour la première fois (nécessite une instrumentation du bytecode lors de la phase de construction).
- ❽ `unique` (optionnel) : génère le DDL d'une contrainte d'unicité pour les colonnes. Permet aussi d'en faire la cible d'une `property-ref`.

- 9 `not-null` (optionnel) : génère le DDL d'une contrainte de nullité pour les colonnes.
- 10 `optimistic-lock` (optionnel - par défaut à `true`) : indique si la mise à jour de cette propriété nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine s'il est nécessaire d'incrémenter un numéro de version quand cette propriété est marquée obsolète (`dirty`).
- 11 `generated` (optionnel - defaults to `never`): specifies that this property value is actually generated by the database. See the discussion of [generated properties](#) for more information.

typename peut être :

1. Le nom d'un type basique Hibernate (par ex : `integer`, `string`, `character`, `date`, `timestamp`, `float`, `binary`, `serializable`, `object`, `blob` etc.).
2. le nom d'une classe Java avec un type basique par défaut (par ex : `int`, `float`, `char`, `java.lang.String`, `java.util.Date`, `java.lang.Integer`, `java.sql.Clob` etc.).
3. Le nom d'une classe Java sérialisable.
4. Le nom d'une classe avec un type personnalisé (par ex : `com.illflow.type.MyCustomType`).

Si vous n'indiquez pas un type, Hibernate utilisera la réflexion sur le nom de la propriété pour tenter de trouver le type Hibernate correct. Hibernate essaiera d'interpréter le nom de la classe retournée par le getter de la propriété en utilisant les règles 2, 3, 4 dans cet ordre. Dans certains cas vous aurez encore besoin de l'attribut `type`. (Par exemple, pour distinguer `Hibernate.DATE` et `Hibernate.TIMESTAMP`, ou pour préciser un type personnalisé).

L'attribut `access` permet de contrôler comment Hibernate accédera à la propriété à l'exécution. Par défaut, Hibernate utilisera les méthodes `set/get`. Si vous indiquez `access="field"`, Hibernate ignorera les `getter/setter` et accédera à la propriété directement en utilisant la réflexion. Vous pouvez spécifier votre propre stratégie d'accès aux propriétés en nommant une classe qui implémente l'interface `org.hibernate.propertyaccess.PropertyAccessor`.

Les propriétés dérivées représentent une fonctionnalité particulièrement intéressante. Ces propriétés sont par définition en lecture seule, la valeur de la propriété est calculée au chargement. Le calcul est déclaré comme une expression SQL, qui se traduit par une sous-requête `SELECT` dans la requête SQL qui charge une instance :

```
<property name="totalPrice"
  formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
            WHERE li.productId = p.productId
            AND li.customerId = customerId
            AND li.orderNumber = orderNumber )"/>
```

Remarquez que vous pouvez référencer la propre table des entités en ne déclarant pas un alias sur une colonne particulière (`customerId` dans l'exemple donné). Notez aussi que vous pouvez utiliser le sous-élément de mappage `<formula>` plutôt que d'utiliser l'attribut si vous le souhaitez.

5.1.12. Plusieurs-à-un

Une association ordinaire vers une autre classe persistante est déclarée en utilisant un élément `many-to-one`. Le modèle relationnel est une association de type plusieurs-à-un : une clé étrangère dans une table référence la ou les clé(s) primaire(s) dans la table cible.

```
<many-to-one
  name="propertyName"
  column="column_name"
  class="ClassName"
  cascade="cascade_style"
  fetch="join|select"
  update="true|false"
  insert="true|false"
  property-ref="propertyNameFromAssociatedClass"
  access="field|property|ClassName"
  unique="true|false"
  not-null="true|false"
  optimistic-lock="true|false"
  lazy="proxy|no-proxy|false"
  not-found="ignore|exception"
  entity-name="EntityName"
  formula="arbitrary SQL expression"
  node="element-name|@attribute-name|element/@attribute|."
  embed-xml="true|false"
  index="index_name"
  unique_key="unique_key_id"
  foreign-key="foreign_key_name"
/>
```

- ❶ `name` : le nom de la propriété.
- ❷ `column` (optionnel) : le nom de la colonne de la clé étrangère. Cela peut être aussi spécifié par un ou des sous-élément(s) `<column>`.
- ❸ `class` (optionnel - par défaut, le type de la propriété déterminé par réflexion) : le nom de la classe associée.
- ❹ `cascade` (optionnel) : indique quelles opérations doivent être cascadées de l'objet parent vers l'objet associé.
- ❺ `fetch` (optionnel - par défaut à `select`) : choisit entre le chargement de type jointure externe (outer-join) ou le chargement par select successifs.
- ❻ `update`, `insert` (optionnel - par défaut à `true`) : indique que les colonnes mappées devraient être incluses dans des SQL `UPDATE` et/ou des déclarations `INSERT`. Mettre les deux à `false`, permet une association pure dérivée dont la valeur est initialisée à partir d'une autre

propriété qui mappe à une ou plusieurs mêmes colonnes, ou par un trigger ou une autre application.

- ⑦ `property-ref` (optionnel) : le nom d'une propriété de la classe associée qui est jointe à cette clé étrangère. Si non-spécifiée, la clé primaire de la classe associée est utilisée.
- ⑧ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ⑨ `unique` (optionnel) : génère le DDL d'une contrainte unique pour la clé étrangère. Permet aussi d'en faire la cible d'une `property-ref`. Cela permet de créer une véritable association un-à-un.
- ⑩ `not-null` (optionnel) : active le DDL d'une contrainte de nullité pour les colonnes de clés étrangères.
- ⑪ `optimistic-lock` (optionnel - par défaut à `true`) : indique si les mise à jour de cette propriété nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine s'il est nécessaire d'incrémenter un numéro de version quand cette propriété est marquée obsolète (`dirty`).
- ⑫ `lazy` (optionnel - par défaut à `proxy`) : par défaut, les associations de point uniques utilisent des proxies. `lazy="no-proxy"` indique que cette propriété doit être chargée en différé au premier accès à la variable d'instance (nécessite une instrumentation du bytecode lors de la phase de construction). `lazy="false"` indique que l'association sera toujours chargée.
- ⑬ `not-found` (optionnel - par défaut = `exception`) : spécifie comment les clés étrangères qui référencent des lignes manquantes seront gérées : `ignore` traitera une ligne manquante comme une association nulle.
- ⑭ `entity-name` (optionnel) : le nom de l'entité de la classe associée.
- ⑮ `formula` (optionnel) : une expression SQL qui définit la valeur pour une clé étrangère calculée.

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are divided into three categories. First, basic operations, which include: `persist`, `merge`, `delete`, `save-update`, `evict`, `replicate`, `lock` and `refresh`; second, special values: `delete-orphan`; and third, all comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See [Section 10.11, « Persistance transitive »](#) for a full explanation. Note that single valued, many-to-one and one-to-one, associations do not support orphan delete.

Une déclaration `many-to-one` typique est aussi simple que :

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

L'attribut `property-ref` devrait être utilisé pour mapper seulement des données provenant d'un ancien système où les clés étrangères font référence à une clé unique de la table associée et qui n'est pas la clé primaire. C'est un cas de mauvaise conception relationnelle. Par exemple, supposez que la classe `Product` ait un numéro de série unique qui n'est pas la clé primaire. L'attribut `unique` contrôle la génération DDL par Hibernate avec l'outil `SchemaExport`.

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Ainsi le mappage pour `OrderItem` peut utiliser :

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

Bien que ce ne soit certainement pas encouragé.

Si la clé unique référencée comprend des propriétés multiples de l'entité associée, vous devez mapper ces propriétés à l'intérieur d'un élément nommé `<properties>`.

Si la clé unique référencée est la propriété d'un composant, vous pouvez spécifier le chemin de propriété :

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

5.1.13. Un-à-un

Une association un-à-un vers une autre classe persistante est déclarée avec l'élément `one-to-one`.

```
<one-to-one
  name="propertyName"
  class="ClassName"
  cascade="cascade_style"
  constrained="true|false"
  fetch="join|select"
  property-ref="propertyNameFromAssociatedClass"
  access="field|property|ClassName"
  formula="any SQL expression"
  lazy="proxy|no-proxy|false"
  entity-name="EntityName"
  node="element-name|@attribute-name|element/@attribute|. "
  embed-xml="true|false"
  foreign-key="foreign_key_name"
/>
```

- ❶ `name` : le nom de la propriété.
- ❷ `class` (optionnel - par défaut, le type de la propriété déterminé par réflexion) : le nom de la classe associée.

- ③ `cascade` (optionnel) : indique quelles opérations doivent être cascadées de l'objet parent vers l'objet associé.
- ④ `constrained` (optionnel) : indique qu'une contrainte de clé étrangère sur la clé primaire de la table mappée référence la table de la classe associée. Cette option affecte l'ordre dans lequel chaque `save()` et chaque `delete()` est cascadié et détermine si l'association peut utiliser un proxy (aussi utilisé par l'outil SchemaExport).
- ⑤ `fetch` (optionnel - par défaut à `select`) : choisit entre le chargement de type jointure externe (outer-join) ou le chargement par select successifs.
- ⑥ `property-ref` (optionnel) : le nom de la propriété de la classe associée qui est jointe à la clé primaire de cette classe. Si ce n'est pas spécifié, la clé primaire de la classe associée est utilisée.
- ⑦ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ⑧ `formula` (optionnel) : presque toutes les associations un-à-un pointent sur la clé primaire de l'entité propriétaire. Dans les rares cas différents, vous devez donner une ou plusieurs autres colonnes ou expression à joindre par une formule SQL . Voir `org.hibernate.test.onetooneformula` pour un exemple.
- ⑨ `lazy` (optionnel - par défaut `proxy`) : par défaut, les associations simples sont soumises à proxy. `lazy="no-proxy"` spécifie que la propriété doit être chargée en différé au premier accès à l'instance. (nécessite l'instrumentation du bytecode à la construction). `lazy="false"` indique que l'association sera toujours chargée agressivement. . *Notez que si `constrained="false"`, l'utilisation de proxy est impossible et Hibernate chargera automatiquement l'association .*
- ⑩ `entity-name` (optionnel) : le nom de l'entité de la classe associée.

Il existe deux types d'associations un-à-un :

- association par clé primaire
- association par clé étrangère unique

Les associations par clé primaire ne nécessitent pas une colonne supplémentaire en table ; si deux lignes sont liées par l'association alors les deux lignes de la table partagent la même valeur de clé primaire. Donc si vous voulez que deux objets soient liés par une association par clé primaire, vous devez faire en sorte qu'on leur assigne la même valeur d'identifiant.

Pour une association par clé primaire, ajoutez les mappages suivants à `Employee` et `Person`, respectivement :

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Maintenant, vous devez faire en sorte que les clés primaires des lignes liées dans les tables `PERSON` et `EMPLOYEE` sont égales. On utilise une stratégie Hibernate spéciale de génération d'identifiants appelée `foreign` :

```
<class name="person" table="PERSON">
  <id name="id" column="PERSON_ID">
    <generator class="foreign">
      <param name="property">
>employee</param>
    </generator>
  </id>
  ...
  <one-to-one name="employee"
    class="Employee"
    constrained="true"/>
</class>
>
```

Une instance fraîchement enregistrée de `Person` se voit alors assignée la même valeur de clé primaire que l'instance de `Employee` référencée par la propriété `employee` de cette `Person`.

Alternativement, une clé étrangère avec contrainte d'unicité de `Employee` vers `Person` peut être indiquée ainsi :

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

Et cette association peut être rendue bidirectionnelle en ajoutant ceci au mappage de `Person` :

```
<one-to-one name="employee" class="Employee" property-ref="person"/>
```

5.1.14. Natural-id

```
<natural-id mutable="true|false"/>
  <property ... />
  <many-to-one ... />
  .....
</natural-id>
>
```

Bien que nous recommandions l'utilisation de clés primaires générées, vous devriez toujours essayer d'identifier des clés métier (naturelles) pour toutes vos entités. Une clé naturelle est une propriété ou une combinaison de propriétés unique et non nulle. Si elle est aussi immuable, c'est encore mieux. Mappez les propriétés de la clé naturelle dans l'élément `<natural-id>`. Hibernate

générera la clé unique nécessaire et les contraintes de non-nullité, et votre mappage s'auto-documentera.

Nous vous recommandons fortement d'implémenter `equals()` et `hashCode()` pour comparer les propriétés clés naturelles de l'entité.

Ce mappage n'est pas destiné à être utilisé avec des entités qui ont des clés naturelles.

- `mutable` (optionnel, par défaut à `false`) : par défaut, les identifiants naturels sont supposés être immuables (constants).

5.1.15. Component, dynamic-component

L'élément `<component>` mappe les propriétés d'un objet enfant aux colonnes d'une classe parente. Les composants peuvent en retour déclarer leurs propres propriétés, composants ou collections. Voir "Components" plus bas :

```
<component
    name="propertyName"
    class="className"
    insert="true|false"
    update="true|false"
    access="field|property|ClassName"
    lazy="true|false"
    optimistic-lock="true|false"
    unique="true|false"
    node="element-name|."
>

    <property ..../>
    <many-to-one .... />
    .....
</component>
>
```

1
2
3
4
5
6
7
8

- 1 `name` : le nom de la propriété.
- 2 `class` (optionnel - par défaut au type de la propriété déterminé par réflexion) : le nom de la classe (enfant) du composant.
- 3 `insert` : les colonnes mappées apparaissent-elles dans les SQL `INSERT` s ?
- 4 `update` : les colonnes mappées apparaissent-elles dans les SQL `UPDATE` s ?
- 5 `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- 6 `lazy` (optionnel - par défaut à `false`) : indique que ce composant doit être chargé en différé au premier accès à la variable d'instance (nécessite une instrumentation du bytecode lors de la phase de construction).

- ⑦ `optimistic-lock` (optionnel - par défaut à `true`) : spécifie si les mise à jour sur ce composant nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine si une incrémentation de version doit avoir lieu quand la propriété est marquée obsolète (`dirty`).
- ⑧ `unique` (optionnel - par défaut à `false`) : Indique qu'une contrainte d'unicité existe sur toutes les colonnes mappées de ce composant.

Les balises enfant `<property>` mappent les propriétés de la classe enfant sur les colonnes de la table.

L'élément `<component>` permet de déclarer un sous-élément `<parent>` qui associe une propriété de la classe composant comme une référence arrière vers l'entité contenante.

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map. See [Section 8.5, « Les composants dynamiques »](#) for more information.

5.1.16. Propriétés

L'élément `<properties>` permet la définition d'un groupement logique nommé des propriétés d'une classe. L'utilisation la plus importante de cette construction est la possibilité pour une combinaison de propriétés d'être la cible d'un `property-ref`. C'est aussi un moyen pratique de définir une contrainte d'unicité multi-colonnes. Par exemple :

```
<properties
    name="logicalName"
    insert="true|false"
    update="true|false"
    optimistic-lock="true|false"
    unique="true|false"
>

    <property ...../>
    <many-to-one .... />
    .....
</properties>
>
```

- ① `name` : le nom logique d'un regroupement et *non* le véritable nom d'une propriété.
- ② `insert` : les colonnes mappées apparaissent-elles dans les SQL `INSERT` s ?
- ③ `update` : les colonnes mappées apparaissent-elles dans les SQL `UPDATE` s ?
- ④ `optimistic-lock` (optionnel - par défaut à `true`) : indique si les mise à jour sur ce composant nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine si une incrémentation de version doit avoir lieu quand la propriété est marquée obsolète (`dirty`).
- ⑤ `unique` (optionnel - par défaut à `false`) : Indique qu'une contrainte d'unicité existe sur toutes les colonnes mappées de ce composant.

Par exemple, si nous avons le mappage de `<properties>` suivant :

```
<class name="Person">
  <id name="personNumber"/>

  ...
  <properties name="name"
    unique="true" update="false">
    <property name="firstName"/>
    <property name="initial"/>
    <property name="lastName"/>
  </properties>
</class>
>
```

Alors nous pourrions avoir une association sur des données d'un ancien système qui font référence à cette clé unique de la table `Person` au lieu de la clé primaire :

```
<many-to-one name="person"
  class="Person" property-ref="name">
  <column name="firstName"/>
  <column name="initial"/>
  <column name="lastName"/>
</many-to-one>
>
```

Nous ne recommandons pas une telle utilisation, en dehors du contexte de mappage de données héritées d'anciens systèmes.

5.1.17. Subclass

Pour finir, la persistance polymorphique nécessite la déclaration de chaque sous-classe de la classe racine persistante. Pour la stratégie de mappage de type `table-per-class-hierarchy`, on utilise la déclaration `<subclass>`.

```
<subclass
  name="ClassName"
  discriminator-value="discriminator_value"
  proxy="ProxyInterface"
  lazy="true|false"
  dynamic-update="true|false"
  dynamic-insert="true|false"
  entity-name="EntityName"
  node="element-name"
  extends="SuperclassName">

  <property .... />
  .....
```

1
2
3
4

```
</subclass>
>
```

- ❶ name : le nom de classe complet de la sous-classe.
- ❷ discriminator-value (optionnel - par défaut le nom de la classe) : une valeur qui distingue les différentes sous-classes.
- ❸ proxy (optionnel) : indique une classe ou interface à utiliser pour l'initialisation différée des proxies.
- ❹ lazy (optionnel, par défaut à true) : spécifier lazy="false" désactive l'utilisation de l'extraction différée.

Chaque sous-classe devrait déclarer ses propres propriétés persistantes et sous-classes. Les propriétés <version> et <id> sont implicitement hérités de la classe racine. Chaque sous-classe dans une hiérarchie doit définir une unique discriminator-value. Si non spécifiée, le nom complet de la classe Java est utilisé.

For information about inheritance mappings see [Chapitre 9, Mapping d'héritage de classe](#).

5.1.18. Joined-subclass

Il est également possible de mapper chaque sous-classe vers sa propre table (stratégie de mappage de type table-per-subclass). L'état hérité est récupéré en joignant la table de la super-classe. L'élément <joined-subclass> est utilisé. Par exemple :

```
<joined-subclass
  name="ClassName"
  table="tablename"
  proxy="ProxyInterface"
  lazy="true|false"
  dynamic-update="true|false"
  dynamic-insert="true|false"
  schema="schema"
  catalog="catalog"
  extends="SuperclassName"
  persister="ClassName"
  subselect="SQL expression"
  entity-name="EntityName"
  node="element-name">

  <key .... >

  <property .... />
  .....
</joined-subclass>
>
```

- ❶
- ❷
- ❸
- ❹

- ❶ name : le nom de classe complet de la sous-classe.
- ❷ table: le nom de la table de la sous-classe.

- ③ proxy (optionnel) : indique une classe ou interface à utiliser pour l'initialisation différée des proxies.
- ④ lazy (optionnel, par défaut à true) : spécifier lazy="false" désactive l'utilisation de l'extraction différée.

Aucune colonne discriminante n'est nécessaire pour cette stratégie de mappage. Cependant, chaque sous-classe doit déclarer une colonne de table contenant l'objet identifiant qui utilise l'élément <key>. Le mappage au début de ce chapitre serait ré-écrit ainsi :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class name="Cat" table="CATS">
        <id name="id" column="uid" type="long">
            <generator class="hilo"/>
        </id>
        <property name="birthdate" type="date"/>
        <property name="color" not-null="true"/>
        <property name="sex" not-null="true"/>
        <property name="weight"/>
        <many-to-one name="mate"/>
        <set name="kittens">
            <key column="MOTHER"/>
            <one-to-many class="Cat"/>
        </set>
        <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
            <key column="CAT"/>
            <property name="name" type="string"/>
        </joined-subclass>
    </class>

    <class name="eg.Dog">
        <!-- mapping for Dog could go here -->
    </class>

</hibernate-mapping>
>
```

For information about inheritance mappings see [Chapitre 9, Mapping d'héritage de classe](#) .

5.1.19. Union-subclass

Une troisième option est de mapper uniquement les classes concrètes d'une hiérarchie d'héritage vers des tables, (stratégie de type table-per-concrete-class) où chaque table définit tous les états persistants de la classe, y compris les états hérités. Dans Hibernate il n'est absolument pas nécessaire de mapper explicitement de telles hiérarchies d'héritage. Vous pouvez simplement mapper chaque classe avec une déclaration <class> différente. Cependant, si vous souhaitez

utiliser des associations polymorphiques (c'est-à-dire une association vers la superclasse de votre hiérarchie), vous devez utiliser le mappage `<union-subclass>`. Par exemple :

```
<union-subclass
    name="ClassName"
    table="tablename"
    proxy="ProxyInterface"
    lazy="true|false"
    dynamic-update="true|false"
    dynamic-insert="true|false"
    schema="schema"
    catalog="catalog"
    extends="SuperclassName"
    abstract="true|false"
    persister="ClassName"
    subselect="SQL expression"
    entity-name="EntityName"
    node="element-name">

    <property .... />
    ....
</union-subclass>
>
```

- ❶ name : le nom de classe complet de la sous-classe.
- ❷ table: le nom de la table de la sous-classe.
- ❸ proxy (optionnel) : indique une classe ou interface à utiliser pour l'initialisation différée des proxies.
- ❹ lazy (optionnel, par défaut à true) : spécifier `lazy="false"` désactive l'utilisation de l'extraction différée.

Aucune colonne discriminante ou colonne clé n'est requise pour cette stratégie de mappage.

For information about inheritance mappings see [Chapitre 9, Mapping d'héritage de classe](#) .

5.1.20. Join

En utilisant l'élément `<join>`, il est possible de mapper des propriétés d'une classe sur plusieurs tables quand il existe une relation un-à-un entre les tables. Par exemple :

```
<join
    table="tablename"
    schema="owner"
    catalog="catalog"
    fetch="join|select"
    inverse="true|false"
    optional="true|false">
```

```
<key ... />

<property ... />
...
</join>
>
```

- ❶ `table` : le nom de la table jointe.
- ❷ `schema` (optionnel) : surcharge le nom de schéma spécifié par l'élément racine `<hibernate-mapping>`.
- ❸ `catalog` (optionnel) : surcharge le nom du catalogue spécifié par l'élément racine `<hibernate-mapping>`.
- ❹ `fetch` (optionnel - par défaut à `join`) : si positionné à `join`, Hibernate utilisera une jointure interne pour charger une `jointure` définie par une classe ou ses super-classes et une jointure externe pour une `<jointure>` définie par une sous-classe. Si positionné à `select`, Hibernate utilisera un `select` séquentiel pour une `<jointure>` définie sur une sous-classe, qui ne sera délivrée que si une ligne représente une instance de la sous-classe. Les jointures internes seront quand même utilisées pour charger une `<jointure>` définie par une classe et ses super-classes.
- ❺ `inverse` (optionnel - par défaut à `false`) : si positionné à `true`, Hibernate n'essaiera pas d'insérer ou de mettre à jour les propriétés définies par cette jointure.
- ❻ `optionnel` (optionnel - par défaut à `false`) : si positionné à `true`, Hibernate insérera une ligne seulement si les propriétés définies par cette jointure sont non-nulles et utilisera toujours une jointure externe pour extraire les propriétés.

Par exemple, les informations d'adresse pour une personne peuvent être mappées vers une table séparée (tout en préservant des sémantiques de type valeur pour toutes ses propriétés) :

```
<class name="Person"
  table="PERSON">

  <id name="id" column="PERSON_ID"
>...</id>

  <join table="ADDRESS">
    <key column="ADDRESS_ID"/>
    <property name="address"/>
    <property name="zip"/>
    <property name="country"/>
  </join>
  ...
```

Cette fonctionnalité est souvent seulement utile pour les modèles de données hérités d'anciens systèmes, nous recommandons d'utiliser moins de tables que de classes et un modèle de domaine à granularité fine. Cependant, c'est utile pour passer d'une stratégie de mappage d'héritage à une autre dans une hiérarchie simple, comme nous le verrons plus tard.

5.1.21. Key

Nous avons rencontré l'élément `<key>` à plusieurs reprises maintenant. Il apparaît partout que l'élément de mappage parent définit une jointure sur une nouvelle table, et définit la clé étrangère dans la table jointe, qui référence la clé primaire de la table d'origine :

```
<key
  column="columnname"
  on-delete="noaction|cascade"
  property-ref="propertyName"
  not-null="true|false"
  update="true|false"
  unique="true|false"
/>
```

- ❶ `column` (optionnel) : le nom de la colonne de la clé étrangère. Cela peut être aussi spécifié par un ou des sous-élément(s) `<column>`.
- ❷ `on-delete` (optionnel, par défaut à `noaction`) : indique si la contrainte de clé étrangère possède la possibilité au niveau base de données de suppression en cascade.
- ❸ `property-ref` (optionnel) : indique que la clé étrangère fait référence à des colonnes qui ne sont pas la clé primaire de la table d'origine (Pour les données d'anciens systèmes).
- ❹ `not-null` (optionnel) : indique que les colonnes des clés étrangères ne peuvent pas être nulles (c'est implicite si la clé étrangère fait partie de la clé primaire).
- ❺ `update` (optionnel) : indique que la clé étrangère ne devrait jamais être mise à jour (implicite si celle-ci fait partie de la clé primaire).
- ❻ `unique` (optionnel) : indique que la clé étrangère doit posséder une contrainte d'unicité (implicite si la clé étrangère est aussi la clé primaire).

Là où les suppressions doivent être performantes, nous recommandons pour les systèmes de définir toutes les clés `on-delete="cascade"`, ainsi Hibernate utilisera une contrainte `ON CASCADE DELETE` au niveau base de données, plutôt que de nombreux `DELETE` individuels. Attention, cette fonctionnalité court-circuite la stratégie habituelle de verrou optimiste pour les données versionnées.

Les attributs `not-null` et `update` sont utiles pour mapper une association un-à-plusieurs unidirectionnelle. Si vous mappez un un-à-plusieurs unidirectionnel vers une clé étrangère non nulle, vous devez déclarer la colonne de la clé en utilisant `<key not-null="true">`.

5.1.22. Éléments `column` et `formula`

Tout élément de mappage qui accepte un attribut `column` acceptera alternativement un sous-élément `<column>`. Pareillement `<formula>` est une alternative à l'attribut `formula`. Par exemple :

```
<column
```

```
name="column_name"
length="N"
precision="N"
scale="N"
not-null="true|false"
unique="true|false"
unique-key="multicolumn_unique_key_name"
index="index_name"
sql-type="sql_type_name"
check="SQL expression"
default="SQL expression"
read="SQL expression"
write="SQL expression"/>
```

```
<formula
>SQL expression</formula
>
```

Most of the attributes on `column` provide a means of tailoring the DDL during automatic schema generation. The `read` and `write` attributes allow you to specify custom SQL that Hibernate will use to access the column's value. For more on this, see the discussion of [column read and write expressions](#).

The `column` and `formula` elements can even be combined within the same property or association mapping to express, for example, exotic join conditions.

```
<many-to-one name="homeAddress" class="Address"
  insert="false" update="false">
  <column name="person_id" not-null="true" length="10"/>
  <formula
>'MAILING'</formula>
</many-to-one
>
```

5.1.23. Import

Supposez que votre application possède deux classes persistantes du même nom, et vous ne voulez pas préciser le nom Java complet (paquetage) dans les requêtes Hibernate. Les classes peuvent alors être "importées" explicitement plutôt que de compter sur `auto-import="true"`. Vous pouvez même importer des classes et interfaces qui ne sont pas mappées explicitement :

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
```



```

class="ClassName"

rename="ShortName"

/>

```

1

2

- 1 class : nom complet de toute classe Java.
- 2 rename (optionnel - par défaut vaut le nom de la classe non qualifié): nom pouvant être utilisé dans le langage de requête.

5.1.24. Any

Il existe encore un type de mappage de propriété. L'élément de mappage `<any>` définit une association polymorphique vers des classes de tables multiples. Ce type de mappage requiert toujours plus d'une colonne. La première colonne contient le type de l'entité associée. Les colonnes restantes contiennent l'identifiant. Il est impossible de spécifier une contrainte de clé étrangère pour ce type d'association, donc ce n'est certainement pas considéré comme le moyen habituel de mapper des associations (polymorphiques). Ne doit être utilisé que dans des cas particuliers (par exemple des logs d'audit, des données de session utilisateur, etc...).

L'attribut `meta-type` permet à l'application de spécifier un type personnalisé qui mappe des valeurs de colonnes de base de données sur des classes persistantes qui ont un attribut identifiant du type spécifié par `id-type`. Vous devez spécifier le mappage à partir de valeurs du méta-type sur les noms des classes.

```

<any name="being" id-type="long" meta-type="string">
  <meta-value value="TBL_ANIMAL" class="Animal" />
  <meta-value value="TBL_HUMAN" class="Human" />
  <meta-value value="TBL_ALIEN" class="Alien" />
  <column name="table_name" />
  <column name="id" />
</any>
>

```

```

<any

  name="propertyName"

  id-type="idtypename"

  meta-type="metatypename"

  cascade="cascade_style"

  access="field|property|ClassName"

  optimistic-lock="true|false"

>

  <meta-value ... />
  <meta-value ... />
  .....
  <column .... />
  <column .... />
  .....

```

1

2

3

4

5

6

```
</any  
>
```

- ❶ `name` : le nom de la propriété.
- ❷ `id-type` : le type identifiant.
- ❸ `meta-type` (optionnel - par défaut à `string`) : tout type permis pour un mappage par discriminateur.
- ❹ `cascade` (optionnel - par défaut à `none`) : le style de cascade.
- ❺ `access` (optionnel - par défaut `property`) : la stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.
- ❻ `optimistic-lock` (optionnel - par défaut à `true`) : indique si les mise à jour sur cette propriété nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, définit si un incrément de version doit avoir lieu quand cette propriété est marquée dirty.

5.2. Types Hibernate

5.2.1. Entités et valeurs

Pour le service de persistance, les objets sont classés en deux groupes au niveau langage Java :

Une *entité* existe indépendamment de tout autre objet possédant des références vers l'entité. Comparez cela avec le modèle Java habituel où un objet est supprimé par le garbage collector dès qu'il n'est plus référencé. Les entités doivent être explicitement enregistrées et supprimées (sauf dans les cas où sauvegardes et suppressions sont *cascadées* d'une entité parent vers ses enfants). C'est différent du modèle ODMG de persistance par atteignabilité - et correspond mieux à la façon dont les objets sont habituellement utilisés dans des grands systèmes. Les entités permettent les références circulaires et partagées. Elles peuvent aussi être versionnées.

L'état persistant d'une entité consiste en des références vers d'autres entités et instances de types *valeurs*. Ces valeurs sont des types primitifs, des collections (et non le contenu d'une collection), des composants de certains objets immuables. Contrairement aux entités, les valeurs (et en particulier les collections et composants) *sont* persistées et supprimées par atteignabilité. Comme les valeurs (et types primitifs) sont persistées et supprimées avec l'entité qui les contient, ils ne peuvent pas posséder leurs propres versions. Les valeurs n'ont pas d'identité indépendantes, ainsi elles ne peuvent pas être partagées par deux entités ou collections.

Jusqu'à présent nous avons utilisé le terme "classe persistante" pour parler d'entités. Nous allons continuer à faire ainsi. Cependant, au sens strict, toutes les classes définies par un utilisateur possédant un état persistant ne sont pas des entités. Un *composant* est une classe définie par un utilisateur avec la sémantique d'une valeur. Une propriété Java de type `java.lang.String` a aussi les caractéristiques d'une valeur. Selon cette définition, nous sommes en mesure de déclarer que tous les types (classes) fournis par JDK possèdent la sémantique d'une valeur dans Java, alors que les types définis par un utilisateur pourront être mappés avec des sémantiques entités ou valeur type. Cette décision est prise par le développeur d'application. Un bon conseil pour une classe entité dans un modèle de domaine sont des références partagées à une instance

unique de cette classe, alors que la composition ou l'agrégation se traduit en général par une valeur type.

Nous nous pencherons sur ces deux concepts tout au long de la documentation.

Le défi est de mapper les types Javas (et la définition des développeurs des entités et valeurs types) sur les types du SQL ou des bases de données. Le pont entre les deux systèmes est proposé par Hibernate : pour les entités nous utilisons `<class>`, `<subclass>` et ainsi de suite. Pour les types valeurs nous utilisons `<property>`, `<component>`, etc., habituellement avec un attribut `type`. La valeur de cet attribut est le nom d'un *type de mappage* Hibernate. Hibernate propose de nombreux mappages prêts à l'utilisation (pour les types de valeurs standards du JDK). Vous pouvez écrire vos propres types de mappages et implémenter aussi vos propres stratégies de conversion comme nous le verrons plus tard.

Tous les types proposés Hibernate à part les collections autorisent les sémantiques null.

5.2.2. Types valeurs de base

Les *types de mappage de base* peuvent être classés de la façon suivante :

`integer`, `long`, `short`, `float`, `double`, `character`, `byte`, `boolean`, `yes_no`, `true_false`

Les mappages de type des primitives Java ou leurs classes wrappers (ex : `Integer` pour `int`) vers les types de colonne SQL (propriétaires) appropriés. `boolean`, `yes_no` et `true_false` sont tous des alternatives pour les types Java `boolean` ou `java.lang.Boolean`.

`string`

Mappage de type de `java.lang.String` vers `VARCHAR` (ou le `VARCHAR2` Oracle).

`date`, `time`, `timestamp`

mappages de type pour `java.util.Date` et ses sous-classes vers les types SQL `DATE`, `TIME` et `TIMESTAMP` (ou équivalent).

`calendar`, `calendar_date`

mappages de type pour `java.util.Calendar` vers les types SQL `TIMESTAMP` et `DATE` (ou équivalent).

`big_decimal`, `big_integer`

mappages de type de `java.math.BigDecimal` et `java.math.BigInteger` vers `NUMERIC` (ou le `NUMBER` Oracle).

`locale`, `timezone`, `currency`

mappages de type pour `java.util.Locale`, `java.util.TimeZone` et `java.util.Currency` vers `VARCHAR` (ou le `VARCHAR2` Oracle). Les instances de `Locale` et `Currency` sont mappées sur leurs codes ISO. Les instances de `TimeZone` sont mappées sur leur ID.

`class`

Un type de mappage de `java.lang.Class` vers `VARCHAR` (ou le `VARCHAR2` Oracle). Un objet `Class` est mappé sur son nom Java complet.

`binary`

Mappe les tableaux de bytes vers le type binaire SQL approprié.

`text`

Mappe les longues chaînes de caractères Java vers les types SQL `CLOB` ou `TEXT`.

`serializable`

Mappe les types Java sérialisables vers le type SQL binaire approprié. Vous pouvez aussi indiquer le type Hibernate `serializable` avec le nom d'une classe Java sérialisable ou une interface qui ne soit pas par défaut un type de base.

`clob`, `blob`

Mappages de type pour les classes JDBC `java.sql.Clob` et `java.sql.Blob`. Ces types peuvent ne pas convenir pour certaines applications car un objet blob ou clob peut ne pas être réutilisable en dehors d'une transaction (de plus l'implémentation par les pilotes comporte des lacunes).

`imm_date`, `imm_time`, `imm_timestamp`, `imm_calendar`, `imm_calendar_date`,
`imm_serializable`, `imm_binary`

Mappages de type pour ceux qui sont habituellement considérés comme des types Java modifiables, et pour lesquels Hibernate effectue certaines optimisations convenant seulement aux types Java immuables. L'application les traite comme immuables. Par exemple, vous ne devriez pas appeler `Date.setTime()` sur une instance mappée sur un `imm_timestamp`. Pour changer la valeur de la propriété, et faire en sorte que cette modification soit persistée, l'application doit assigner un nouvel (non identique) objet à la propriété.

Les identifiants uniques des entités et collections peuvent être de n'importe quel type de base excepté `binary`, `blob` et `clob` (les identifiants composites sont aussi permis, voir plus bas).

Les types de base des valeurs ont des `Type` constants correspondants et définis dans `org.hibernate.Hibernate`. Par exemple, `Hibernate.STRING` représente le type `string`.

5.2.3. Types de valeur personnalisés

Il est assez facile pour les développeurs de créer leurs propres types de valeurs. Par exemple, vous aimeriez persister des propriétés du type `java.lang.BigInteger` dans des colonnes `VARCHAR`. Hibernate ne procure pas de type par défaut à cet effet. Toutefois, les types personnalisés ne se limitent pas à mapper des propriétés (ou élément collection) à une simple colonne de table. Donc, par exemple, vous pourriez avoir une propriété Java `getName()/setName()` de type `java.lang.String` persistée dans les colonnes `FIRST_NAME`, `INITIAL`, `SURNAME`.

Pour implémenter votre propre type, vous pouvez soit implémenter `org.hibernate.UserType` soit `org.hibernate.CompositeUserType` et déclarer des propriétés utilisant des noms de classes complets du type. Consultez `org.hibernate.test.DoubleStringType` pour étudier les possibilités.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
```

```

    <column name="first_string"/>
    <column name="second_string"/>
  </property>
>

```

Remarquez l'utilisation des balises `<column>` pour mapper une propriété sur des colonnes multiples.

Les interfaces `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, et `UserVersionType` prennent en charge des utilisations plus spécialisées.

Vous pouvez même fournir des paramètres en indiquant `UserType` dans le fichier de mappage. À cet effet, votre `UserType` doit implémenter l'interface `org.hibernate.usertype.ParameterizedType`. Pour spécifier des paramètres dans votre type propre, vous pouvez utiliser l'élément `<type>` dans vos fichiers de mappage.

```

<property name="priority">
  <type name="com.mycompany.usertypes.DefaultValueIntegerType">
    <param name="default">
>0</param>
  </type>
</property>
>

```

Le `UserType` permet maintenant de récupérer la valeur pour le paramètre nommé `default` à partir de l'objet `Properties` qui lui est passé.

Si vous utilisez fréquemment un `UserType`, il est utile de lui définir un nom plus court. Vous pouvez l'effectuer, en utilisant l'élément `<typedef>`. Les `typedefs` permettent d'assigner un nom à votre type propre et peuvent aussi contenir une liste de valeurs de paramètres par défaut si ce type est paramétré.

```

<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
  <param name="default">
>0</param>
</typedef>
>

```

```

<property name="priority" type="default_zero"/>

```

Il est également possible de redéfinir les paramètres par défaut du `typedef` au cas par cas en utilisant des paramètres type sur le mappage de la propriété.

Alors que Hibernate offre une riche variété de types, et la prise en charge des composants, vous aurez très rarement *besoin* d'utiliser un type personnalisé, il est néanmoins recommandé d'utiliser des types personnalisés pour les classes (non entités) qui apparaissent fréquemment

dans votre application. Par exemple, une classe `MonetaryAmount` est un bon candidat pour un `CompositeUserType` même si elle pourrait facilement être mappée en tant que composant. Une motivation pour cela est l'abstraction. Avec un type personnalisé, vos documents de mappage sont à l'abri des changements futurs dans votre façon de représenter des valeurs monétaires.

5.3. Mapper une classe plus d'une fois

Il est possible de fournir plus d'un mappage par classe persistante. Dans ce cas, vous devez spécifier un *nom d'entité* pour lever l'ambiguïté entre les instances des entités mappées (par défaut, le nom de l'entité est celui de la classe). Hibernate vous permet de spécifier le nom de l'entité lorsque vous utilisez des objets persistants, lorsque vous écrivez des requêtes ou quand vous mappez des associations vers les entités nommées.

```
<class name="Contract" table="Contracts"
  entity-name="CurrentContract">
  ...
  <set name="history" inverse="true"
    order-by="effectiveEndDate desc">
    <key column="currentContractId"/>
    <one-to-many entity-name="HistoricalContract"/>
  </set>
</class>

<class name="Contract" table="ContractHistory"
  entity-name="HistoricalContract">
  ...
  <many-to-one name="currentContract"
    column="currentContractId"
    entity-name="CurrentContract"/>
</class>
>
```

Remarquez comment les associations sont désormais spécifiées en utilisant `entity-name` au lieu de `class`.

5.4. SQL quoted identifiers

Vous pouvez forcer Hibernate à mettre un identifiant entre quotes dans le SQL généré en mettant le nom de la table ou de la colonne entre backticks dans le document de mappage. Hibernate utilisera les bons styles de quotes pour le SQL `Dialect` (habituellement des doubles quotes, mais des parenthèses pour SQL Server et des backticks pour MySQL).

```
<class name="LineItem" table="`Line Item`">
  <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
  <property name="itemNumber" column="`Item #`"/>
  ...
</class>
>
```

5.5. Métadonnées alternatives

XML ne convient pas à tout le monde, il y a donc des moyens alternatifs pour définir des métadonnées de mappage O/R dans Hibernate.

5.5.1. Utilisation de XDoclet

De nombreux utilisateurs de Hibernate préfèrent embarquer les informations de mappages directement au sein du code source en utilisant les balises XDoclet `@hibernate.tags`. Nous ne couvrons pas cette approche dans ce document puisque cela est considéré comme faisant partie de XDoclet. Cependant, nous présentons l'exemple suivant de la classe `Cat` avec des mappages XDoclet :

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

/**
 * @hibernate.class
 * table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /**
     * @hibernate.id
     * generator-class="native"
     * column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     * column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
```

```
    * column="BIRTH_DATE"
    */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     * column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     * column="COLOR"
     * not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     * inverse="true"
     * order-by="BIRTH_DATE"
     * @hibernate.collection-key
     * column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     * column="SEX"
     * not-null="true"
     * update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
```



```

        this.sex=sex;
    }
}

```

Voyez le site web de Hibernate pour plus d'exemples sur XDoclet et Hibernate.

5.5.2. Utilisation des annotations JDK 5.0

Le JDK 5.0 introduit des annotations proches de celles de XDoclet au niveau java, qui sont type-safe et vérifiées à la compilation. Ce mécanisme est plus puissant que XDoclet et mieux supporté par les outils et les IDE. IntelliJ IDEA, par exemple, supporte l'auto-complétion et le surlignement syntaxique des annotations JDK 5.0. La nouvelle révision des spécifications des EJB (JSR-220) utilise les annotations JDK 5.0 comme mécanisme primaire pour les métadonnées des beans entités. Hibernate3 implémente l'*EntityManager* de la JSR-220 (API de persistance), le support du mappage de métadonnées est disponible via le paquetage *Hibernate Annotations*, en tant que module séparé à télécharger. EJB3 (JSR-220) et les métadonnées Hibernate3 sont supportés.

Ceci est un exemple d'une classe POJO annotée comme un EJB entité :

```

@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order
> orders;

    // Getter/setter and business methods
}

```



Note

Notez que le support des annotations JDK 5.0 (et de la JSR-220) est encore en cours et n'est pas terminé. Référez vous au module Hibernate Annotation pour plus d'informations.

5.6. Propriétés générées

Les propriétés générées sont des propriétés dont les valeurs sont générées par la base de données. Typiquement, les applications Hibernate avaient besoin d'invoquer `refresh` sur les instances qui contenaient des propriétés pour lesquelles la base de données générerait des valeurs. Marquer les propriétés comme générées, permet à l'application de déléguer cette responsabilité à Hibernate. Principalement, à chaque fois que Hibernate réalise un SQL INSERT ou UPDATE en base de données pour une entité marquée comme telle, cela provoque immédiatement un select pour récupérer les valeurs générées.

Properties marked as generated must additionally be non-insertable and non-updateable. Only *versions*, *timestamps*, and *simple properties*, can be marked as generated.

`never` (par défaut) - indique que la valeur donnée de la propriété n'est pas générée dans la base de données.

`insert`: the given property value is generated on insert, but is not regenerated on subsequent updates. Properties like created-date fall into this category. Even though *version* and *timestamp* properties can be marked as generated, this option is not available.

`always` - indique que la valeur de la propriété est générée à l'insertion comme aux mise à jour.

5.7. Column read and write expressions

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to *simple properties*. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like this:

```
<property name="creditCardNumber">
  <column
    name="credit_card_num"
    read="decrypt(credit_card_num)"
    write="encrypt(?)" />
</property>
>
```

Hibernate applies the custom expressions automatically whenever the property is referenced in a query. This functionality is similar to a derived-property *formula* with two differences:

- The property is backed by one or more columns that are exported as part of automatic schema generation.
- The property is read-write, not read-only.

The `write` expression, if specified, must contain exactly one '?' placeholder for the value.

5.8. Objets auxiliaires de la base de données

Permettent les ordres CREATE et DROP d'objets arbitraire de la base de données, en conjonction avec les outils Hibernate d'évolutions de schéma, pour permettre de définir complètement un schéma utilisateur au sein des fichiers de mappage Hibernate. Bien que conçu spécifiquement pour créer et supprimer des objets tels que les triggers et les procédures stockées, en réalité toute commande pouvant être exécutée via une méthode de `java.sql.Statement.execute()` (ALTERs, INSERTS, etc) est valable à cet endroit. Il y a principalement deux modes pour définir les objets auxiliaires de base de données :

Le premier mode est de lister explicitement les commandes CREATE et DROP dans le fichier de mappage :

```
<hibernate-mapping>
...
  <database-object>
    <create
>CREATE TRIGGER my_trigger ...</create>
    <drop
>DROP TRIGGER my_trigger</drop>
  </database-object>
</hibernate-mapping>
>
```

Le second mode est de fournir une classe personnalisée qui sait comment construire les commandes CREATE et DROP. Cette classe personnalisée doit implémenter l'interface `org.hibernate.mapping.AuxiliaryDatabaseObject`.

```
<hibernate-mapping>
...
  <database-object>
    <definition class="MyTriggerDefinition"/>
  </database-object>
</hibernate-mapping>
>
```

De plus, ces objets de base de données peuvent être optionnellement traités selon l'utilisation de dialectes particuliers.

```
<hibernate-mapping>
...
  <database-object>
    <definition class="MyTriggerDefinition"/>
    <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
    <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
  </database-object>
</hibernate-mapping>
```

>

Mapper une collection

6.1. Collections persistantes

Hibernate requiert que les champs contenant des collections persistantes soient déclarés comme des types d'interface, par exemple :

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

L'interface réelle peut être `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` ou n'importe quoi d'autre ! (Où "n'importe quoi d'autre" signifie que vous devrez écrire une implémentation de `org.hibernate.usertype.UserCollectionType`.)

Notez comment nous avons initialisé la variable d'instance avec une instance de `HashSet`. C'est le meilleur moyen pour initialiser les collections d'instances nouvellement créées (non persistantes). Quand nous fabriquons l'instance persistante - en appelant `persist()`, par exemple - Hibernate remplacera réellement le `HashSet` par une instance d'une implémentation propre à Hibernate de `Set`. Prenez garde aux erreurs suivantes :

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

Les collections persistantes injectées par Hibernate se comportent de la même manière que `HashMap`, `HashSet`, `TreeMap`, `TreeSet` ou `ArrayList`, selon le type de l'interface.

Les instances des collections ont le comportement habituel des types de valeurs. Elles sont automatiquement persistées quand elles sont référencées par un objet persistant et automatiquement effacées quand elles sont déréférencées. Si une collection est passée d'un objet persistant à un autre, ses éléments peuvent être déplacés d'une table à une autre. Deux entités ne peuvent pas partager une référence vers une même instance de collection. Dû au modèle

relationnel sous-jacent, les propriétés contenant des collections ne supportent pas la sémantique de la valeur null ; Hibernate ne fait pas de distinction entre une référence de collection nulle et une collection vide.

Ne vous en souciez pas trop. Utilisez les collections persistantes de la même manière que vous utilisez des collections Java ordinaires. Assurez-vous de comprendre la sémantique des associations bidirectionnelles (traitée plus loin).

6.2. Mapper une collection



Astuce

Il y a une grande variété de mappages qui peuvent être générés pour les collections, couvrant beaucoup de nombreux modèles relationnels communs. Nous vous suggérons d'expérimenter avec l'outil de génération de schéma pour cerner comment les différentes déclarations de mappage se traduisent vers des tables de bases de données.

L'élément de mappage d'Hibernate utilisé pour mapper une collection dépend du type de l'interface. Par exemple, un élément `<set>` est utilisé pour mapper des propriétés de type `Set`.

```
<class name="Product">
  <id name="serialNumber" column="productSerialNumber"/>
  <set name="parts">
    <key column="productSerialNumber" not-null="true"/>
    <one-to-many class="Part"/>
  </set>
</class>
>
```

À part `<set>`, il y aussi les éléments de mappage `<list>`, `<map>`, `<bag>`, `<array>` et `<primitive-array>`. L'élément `<map>` est représentatif :

```
<map
  name="propertyName"
  table="table_name"
  schema="schema_name"
  lazy="true|extra|false"
  inverse="true|false"
  cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"
  sort="unsorted|natural|comparatorClass"
  order-by="column_name asc|desc"
  where="arbitrary sql where condition"
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

```

    fetch="join|select|subselect"
    batch-size="N"
    access="field|property|ClassName"
    optimistic-lock="true|false"
    mutable="true|false"
    node="element-name|."
    embed-xml="true|false"
  >

  <key .... />
  <map-key .... />
  <element .... />
</map>
>

```

- ❶ `name` : le nom de la propriété contenant la collection
- ❷ `table` (optionnel - par défaut = nom de la propriété) : le nom de la table de la collection (non utilisé pour les associations un-à-plusieurs)
- ❸ `schema` (optionnel) : le nom du schéma pour surcharger le schéma déclaré dans l'élément racine
- ❹ `lazy` (optionnel - par défaut = `true`) : peut être utilisé pour désactiver l'initialisation tardive et spécifier que l'association est toujours rapportée, ou pour activer la récupération extra-paresseuse (extra-lazy) où la plupart des opérations n'initialisent pas la collection (approprié pour de très grosses collections).
- ❺ `inverse` (optionnel - par défaut = `false`) : définit cette collection comme l'extrémité "inverse" de l'association bidirectionnelle.
- ❻ `cascade` (optionnel - par défaut = `none`) : active les opérations de cascade vers les entités filles.
- ❼ `sort` (optionnel) : spécifie une collection triée via un ordre de tri naturel, ou via une classe comparateur donnée.
- ❽ `order-by` (optionnel, seulement à partir du JDK1.4) : spécifie une colonne de table (ou des colonnes) qui définit l'ordre d'itération de `Map`, `Set` ou `Bag`, avec en option `asc` ou `desc`.
- ❾ `where` (optionnel) : spécifie une condition SQL arbitraire `WHERE` à utiliser au chargement ou à la suppression d'une collection (utile si la collection ne doit contenir qu'un sous ensemble des données disponibles).
- ❿ `fetch` (optionnel, par défaut = `select`) : à choisir entre récupération par jointures externes, récupération par selects séquentiels, et récupération par sous-selects séquentiels.
- ⓫ `batch-size` (optionnel, par défaut = 1) : une "taille de batch" utilisée pour charger plusieurs instances de cette collection.
- ⓬ `access` (optionnel - par défaut = `property`) : la stratégie que Hibernate doit utiliser pour accéder à la valeur de la propriété.
- ⓭ `optimistic-lock` (optionnel - par défaut = `true`) : spécifie que changer l'état des résultats de la collection entraîne l'incrément de la version appartenant à l'entité (Pour une association un-à-plusieurs, il est souvent raisonnable de désactiver ce paramètre).

- 14 `mutable` (optionnel - par défaut = `true`) : une valeur à `false` spécifie que les éléments de la collection ne changent jamais (une optimisation mineure dans certains cas).

6.2.1. Les clés étrangères d'une collection

Les instances d'une collection sont distinguées dans la base de données par la clé étrangère de l'entité qui possède la collection. Cette clé étrangère est référencée comme la(es) *colonne(s) de la clé de la collection* de la table de la collection. La colonne de la clé de la collection est mappée par l'élément `<key>`.

Il peut y avoir une contrainte de nullité sur la colonne de la clé étrangère. Pour les associations unidirectionnelles un-à-plusieurs, la colonne de la clé étrangère peut être nulle par défaut, donc vous pourriez avoir besoin de spécifier `not-null="true"`.

```
<key column="productSerialNumber" not-null="true" />
```

La contrainte de la clé étrangère peut utiliser `ON DELETE CASCADE`.

```
<key column="productSerialNumber" on-delete="cascade" />
```

Voir le chapitre précédent pour une définition complète de l'élément `<key>`.

6.2.2. Les éléments d'une collection

Les collections peuvent contenir la plupart des autres types Hibernate, y compris tous les types basiques, les types utilisateur, les composants, et bien sûr, les références vers d'autres entités. C'est une distinction importante. Un objet dans une collection pourrait être géré avec une sémantique de "valeur" (sa durée de vie dépend complètement du propriétaire de la collection) ou il pourrait avoir une référence vers une autre entité, avec sa propre durée de vie. Dans le dernier cas, seul le "lien" entre les deux objets est considéré être l'état retenu par la collection.

Le type contenu est référencé comme le *type de l'élément de la collection*. Les éléments de la collections sont mappés par `<element>` ou `<composite-element>`, ou dans le cas des références d'entité, avec `<one-to-many>` ou `<many-to-many>`. Les deux premiers mappent des éléments avec une sémantique de valeur, les deux suivants sont utilisés pour mapper des associations d'entité.

6.2.3. Collections indexées

Tous les mappages de collection, exceptés ceux avec les sémantiques d'ensemble (set) et de sac (bag), ont besoin d'une *colonne d'index* dans la table de la collection - une colonne qui mappe un index de tableau, ou un index de `List`, ou une clé de `Map`. L'index d'une `Map` peut être n'importe quel type basique, mappé avec `<map-key>`, ou peut être une référence d'entité mappée avec `<map-key-many-to-many>`, ou peut être un type composé, mappé avec `<composite-map-key>`.

L'index d'un tableau ou d'une liste est toujours de type `integer` et est mappé en utilisant l'élément `<list-index>`. Les colonnes mappées contiennent des entiers séquentiels (numérotés à partir de zéro par défaut).

```
<list-index
  column="column_name"
  base="0|1|..." />
```

①

- ① `column_name` (required): the name of the column holding the collection index values.
- ① `base` (optional - defaults to 0): the value of the index column that corresponds to the first element of the list or array.

```
<map-key
  column="column_name"
  formula="any SQL expression"
  type="type_name"
  node="@attribute-name"
  length="N" />
```

①

②

③

- ① `column` (optional): the name of the column holding the collection index values.
- ② `formula` (optional): a SQL formula used to evaluate the key of the map.
- ③ `type` (required): the type of the map keys.

```
<map-key-many-to-many
  column="column_name"
  formula="any SQL expression"
  class="ClassName"
/>
```

①

② ③

- ① `column` (optional): the name of the foreign key column for the collection index values.
- ② `formula` (optional): a SQ formula used to evaluate the foreign key of the map key.
- ③ `class` (required): the entity class used as the map key.

Si votre table n'a pas de colonne d'index, et que vous souhaitez tout de même utiliser `List` comme type de propriété, vous devriez mapper la propriété comme un `<bag>` Hibernate. Un sac (bag) ne garde pas son ordre quand il est récupéré de la base de données, mais il peut être optionnellement trié ou ordonné.

6.2.4. Collections de valeurs et associations plusieurs-à-plusieurs

Toute collection de valeurs ou association plusieurs-à-plusieurs requiert une *table de collection* avec une(des) colonne(s) de clé étrangère, une(des) *colonne(s) d'élément de la collection* ou des colonnes et éventuellement une(des) colonne(s) d'index.

Pour une collection de valeurs, nous utilisons la balise `<element>`. Par exemple :

```
<element
    column="column_name"
    formula="any SQL expression"
    type="typename"
    length="L"
    precision="P"
    scale="S"
    not-null="true|false"
    unique="true|false"
    node="element-name"
/>
```

- ❶ `column` (optional): the name of the column holding the collection element values.
- ❷ `formula` (optional): an SQL formula used to evaluate the element.
- ❸ `type` (required): the type of the collection element.

A *many-to-many* association is specified using the `<many-to-many>` element.

```
<many-to-many
    column="column_name"
    formula="any SQL expression"
    class="ClassName"
    fetch="select|join"
    unique="true|false"
    not-found="ignore|exception"
    entity-name="EntityName"
    property-ref="propertyNameFromAssociatedClass"
    node="element-name"
    embed-xml="true|false"
/>
```

- ❶ `column` (optional): the name of the element foreign key column.
- ❷ `formula` (optional): an SQL formula used to evaluate the element foreign key value.
- ❸ `class` (required): the name of the associated class.

- ④ `fetch` (optional - defaults to `join`): enables outer-join or sequential select fetching for this association. This is a special case; for full eager fetching in a single `SELECT` of an entity and its many-to-many relationships to other entities, you would enable `join` fetching, not only of the collection itself, but also with this attribute on the `<many-to-many>` nested element.
- ⑤ `unique` (optional): enables the DDL generation of a unique constraint for the foreign-key column. This makes the association multiplicity effectively one-to-many.
- ⑥ `not-found` (optional - defaults to `exception`): specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.
- ⑦ `entity-name` (optional): the entity name of the associated class, as an alternative to `class`.
- ⑧ `property-ref` (optional): the name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used.

Voici quelques exemples :

Un ensemble de chaînes de caractères :

```
<set name="names" table="person_names">
  <key column="person_id"/>
  <element column="person_name" type="string"/>
</set>
>
```

Un sac contenant des entiers (avec un ordre d'itération déterminé par l'attribut `order-by`) :

```
<bag name="sizes"
      table="item_sizes"
      order-by="size asc">
  <key column="item_id"/>
  <element column="size" type="integer"/>
</bag>
>
```

Un tableau d'entités - dans ce cas, une association plusieurs-à-plusieurs :

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
  <key column="personId"/>
  <list-index column="sortOrder"/>
  <many-to-many column="addressId" class="Address"/>
</array>
>
```

Une map de chaînes de caractères vers des dates :

```
<map name="holidays"
```

```
    table="holidays"
    schema="dbo"
    order-by="hol_name asc">
<key column="id"/>
<map-key column="hol_name" type="string"/>
<element column="hol_date" type="date"/>
</map>
>
```

Une liste de composants (traité dans le prochain chapitre) :

```
<list name="carComponents"
    table="CarComponents">
  <key column="carId"/>
  <list-index column="sortOrder"/>
  <composite-element class="CarComponent">
    <property name="price"/>
    <property name="type"/>
    <property name="serialNumber" column="serialNum"/>
  </composite-element>
</list>
>
```

6.2.5. Associations un-à-plusieurs

Une *association un-à-plusieurs* lie les tables de deux classes par une clé étrangère, sans l'intervention d'une table de collection. Ce mappage perd certaines sémantiques des collections Java normales :

- Une instance de la classe de l'entité contenue ne peut pas appartenir à plus d'une instance de la collection.
- Une instance de la classe de l'entité contenue peut ne pas apparaître à plus d'une valeur d'index de la collection.

Une association de `Product` vers `Part` requiert l'existence d'une clé étrangère et éventuellement une colonne d'index pour la table `Part`. Une balise `<one-to-many>` indique que c'est une association un-à-plusieurs.

```
<one-to-many
  class="ClassName"
  not-found="ignore|exception"
  entity-name="EntityName"
  node="element-name"
  embed-xml="true|false"
/>
```

1
2
3

- ❶ `class` (requis) : le nom de la classe associée.

- ② `not-found` (optionnel - par défaut `exception`) : spécifie comment les identifiants cachés qui référencent des lignes manquantes seront gérés : `ignore` traitera une ligne manquante comme une association nulle.
- ③ `entity-name` (optionnel) : le nom de l'entité de la classe associée, comme une alternative à `class`.

Notez que l'élément `<one-to-many>` n'a pas besoin de déclarer de colonnes. Il n'est pas non plus nécessaire de spécifier le nom de la `table` à aucun endroit.



Avertissement

Note très importante : si la colonne de la clé d'une association `<one-to-many>` est déclarée `NOT NULL`, vous devez déclarer le mappage de `<key>` avec `not-null="true"` ou *utiliser une association bidirectionnelle* avec le mappage de la collection marqué `inverse="true"`. Voir la discussion sur les associations bidirectionnelles plus tard dans ce chapitre.

Cet exemple montre une map d'entités `Part` par nom (où `partName` est une propriété persistante de `Part`). Notez l'utilisation d'un index basé sur une formule :

```
<map name="parts"
      cascade="all">
  <key column="productId" not-null="true"/>
  <map-key formula="partName"/>
  <one-to-many class="Part"/>
</map>
>
```

6.3. Mappages de collection avancés

6.3.1. Collections triées

Hibernate supporte des collections implémentant `java.util.SortedMap` et `java.util.SortedSet`. Vous devez spécifier un comparateur dans le fichier de mappage :

```
<set name="aliases"
      table="person_aliases"
      sort="natural">
  <key column="person"/>
  <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
  <key column="year_id"/>
  <map-key column="hol_name" type="string"/>
  <element column="hol_date" type="date"/>
</map>
```

>

Les valeurs permises pour l'attribut `sort` sont `unsorted`, `natural` et le nom d'une classe implémentant `java.util.Comparator`.

Les collections triées se comportent réellement comme `java.util.TreeSet` ou `java.util.TreeMap`.

Si vous voulez que la base de données elle-même ordonne les éléments de la collection, utilisez l'attribut `order-by` des mappages `set`, `bag` ou `map`. Cette solution est seulement disponible à partir du JDK 1.4 (c'est implémenté en utilisant `LinkedHashSet` ou `LinkedHashMap`). Ceci exécute le tri dans la requête SQL, pas en mémoire.

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
  <key column="person"/>
  <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
  <key column="year_id"/>
  <map-key column="hol_name" type="string"/>
  <element column="hol_date" type="date"/>
</map>
>
```



Note

Notez que la valeur de l'attribut `order-by` est un ordre SQL, et non pas un ordre HQL.

Les associations peuvent même être triées sur des critères arbitraires à l'exécution en utilisant un `filter()` de collection :

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

6.3.2. Associations bidirectionnelles

Une *association bidirectionnelle* permet la navigation à partir des deux extrémités de l'association. Deux types d'associations bidirectionnelles sont supportées :

un-à-plusieurs (one-to-many)

ensemble ou sac à une extrémité, une seule valeur à l'autre

plusieurs-à-plusieurs

ensemble ou sac aux deux extrémités

Vous pouvez spécifier une association bidirectionnelle plusieurs-à-plusieurs simplement en mappant deux associations plusieurs-à-plusieurs vers la même table de base de données et en déclarant une extrémité comme *inverse* (celle de votre choix, mais pas une collection indexée).

Voici un exemple d'association bidirectionnelle plusieurs-à-plusieurs ; chaque catégorie peut avoir plusieurs objets et chaque objet peut être dans plusieurs catégories :

```
<class name="Category">
  <id name="id" column="CATEGORY_ID" />
  ...
  <bag name="items" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID" />
    <many-to-many class="Item" column="ITEM_ID" />
  </bag>
</class>

<class name="Item">
  <id name="id" column="ITEM_ID" />
  ...

  <!-- inverse end -->
  <bag name="categories" table="CATEGORY_ITEM" inverse="true">
    <key column="ITEM_ID" />
    <many-to-many class="Category" column="CATEGORY_ID" />
  </bag>
</class>
>
```

Les changements faits uniquement sur l'extrémité inverse de l'association *ne sont pas* persistés. Ceci signifie qu'Hibernate a deux représentations en mémoire pour chaque association bidirectionnelle, un lien de A vers B et un autre de B vers A. Ceci est plus facile à comprendre si vous pensez au modèle objet de Java et à la façon dont nous créons une relation plusieurs-à-plusieurs dans Java :

```
category.getItems().add(item);           // The category now "knows" about the relationship
item.getCategories().add(category);       // The item now "knows" about the relationship

session.persist(item);                    // The relationship won't be saved!
session.persist(category);                 // The relationship will be saved
```

La partie non-inverse est utilisée pour sauvegarder la représentation en mémoire dans la base de données.

Vous pouvez définir une association bidirectionnelle un-à-plusieurs en mappant une association un-à-plusieurs vers la(es) même(s) colonne(s) de table qu'une association plusieurs-à-un et en déclarant l'extrémité pluri-valuée *inverse="true"*.

```
<class name="Parent">
```

```
<id name="id" column="parent_id"/>
....
<set name="children" inverse="true">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
</class>

<class name="Child">
  <id name="id" column="child_id"/>
  ....
  <many-to-one name="parent"
    class="Parent"
    column="parent_id"
    not-null="true"/>
</class>
>
```

Mapper une extrémité d'une association avec `inverse="true"` n'affecte pas l'opération de cascades, ce sont des concepts orthogonaux.

6.3.3. Associations bidirectionnelles avec des collections indexées

Une association bidirectionnelle où une extrémité est représentée comme une `<list>` ou une `<map>` requiert une considération spéciale. S'il y a une propriété de la classe enfant qui mappe la colonne de l'index, pas de problème, nous pouvons continuer à utiliser `inverse="true"` sur le mappage de la collection :

```
<class name="Parent">
  <id name="id" column="parent_id"/>
  ....
  <map name="children" inverse="true">
    <key column="parent_id"/>
    <map-key column="name"
      type="string"/>
    <one-to-many class="Child"/>
  </map>
</class>

<class name="Child">
  <id name="id" column="child_id"/>
  ....
  <property name="name"
    not-null="true"/>
  <many-to-one name="parent"
    class="Parent"
    column="parent_id"
    not-null="true"/>
</class>
>
```


Mais, si il n'y a pas de telle propriété sur la classe enfant, nous ne pouvons pas considérer l'association comme vraiment bidirectionnelle (il y a des informations disponibles à une extrémité de l'association qui ne sont pas disponibles à l'autre extrémité). Dans ce cas, nous ne pouvons pas mapper la collection `inverse="true"`. Par contre, nous utiliserons le mappage suivant :

```
<class name="Parent">
  <id name="id" column="parent_id"/>
  ....
  <map name="children">
    <key column="parent_id"
      not-null="true"/>
    <map-key column="name"
      type="string"/>
    <one-to-many class="Child"/>
  </map>
</class>

<class name="Child">
  <id name="id" column="child_id"/>
  ....
  <many-to-one name="parent"
    class="Parent"
    column="parent_id"
    insert="false"
    update="false"
    not-null="true"/>
</class>
>
```

Note that in this mapping, the collection-valued end of the association is responsible for updates to the foreign key.

6.3.4. Associations ternaires

Il y a trois approches possibles pour mapper une association ternaire. L'une est d'utiliser une `Map` avec une association comme son index :

```
<map name="contracts">
  <key column="employer_id" not-null="true"/>
  <map-key-many-to-many column="employee_id" class="Employee"/>
  <one-to-many class="Contract"/>
</map>
>
```

```
<map name="connections">
  <key column="incoming_node_id"/>
  <map-key-many-to-many column="outgoing_node_id" class="Node"/>
  <many-to-many column="connection_id" class="Connection"/>
</map>
```

>

Une seconde approche est simplement de remodeler l'association comme une classe d'entité. C'est l'approche la plus commune.

Une alternative finale est d'utiliser des éléments composites, dont nous discuterons plus tard.

6.3.5. Using an `<idbag>`

Si vous êtes bien d'accord avec nous sur le fait que les clés composées sont une mauvaise chose et que les entités devraient avoir des identifiants artificiels (des clés subrogées), vous pourrez trouver un peu curieux que les associations plusieurs-à-plusieurs et les collections de valeurs que nous avons montrées jusqu'ici, mappent toutes des tables avec des clés composées ! Il est vrai que ce point est ambigu ; une table d'association pure ne semble pas tirer avantage d'une clé subrogée (bien qu'une collection de valeur composées le *pourrait*). Néanmoins, Hibernate fournit une fonctionnalité qui vous permet de mapper des associations plusieurs-à-plusieurs et des collections de valeurs vers une table avec une clé subrogée.

L'élément `<idbag>` vous laisse mapper une `List` (ou une `Collection`) avec une sémantique de sac. Par exemple :

```
<idbag name="lovers" table="LOVERS">
  <collection-id column="ID" type="long">
    <generator class="sequence"/>
  </collection-id>
  <key column="PERSON1"/>
  <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
>
```

Comme vous pouvez le constater, un `<idbag>` a un générateur d'id artificiel, exactement comme une classe d'entité ! Une clé subrogée différente est assignée à chaque ligne de la collection. Cependant, Hibernate ne fournit pas de mécanisme pour découvrir la valeur d'une clé subrogée d'une ligne particulière.

Notez que les performances de la mise à jour d'un `<idbag>` sont *bien* meilleures qu'un `<bag>` ordinaire ! Hibernate peut localiser des lignes individuelles efficacement et les mettre à jour ou les effacer individuellement, comme une liste, une map ou un ensemble.

Dans l'implémentation actuelle, la stratégie de la génération de l'identifiant `native` n'est pas supportée pour les identifiants de collection `<idbag>`.

6.4. Exemples de collections

Exemples de collections

La classe suivante possède une collection d'instances `Child`(filles) :

```

package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}

```

Si chaque instance fille a au plus un parent, le mappage le plus naturel est une association un-à-plusieurs :

```

<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children">
      <key column="parent_id"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
>

```

Ceci mappe les définitions de tables suivantes :

```

create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent

```

Si le parent est *requis*, utilisez une association bidirectionnelle un-à-plusieurs :

```
<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" inverse="true">
      <key column="parent_id"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
    <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
  </class>

</hibernate-mapping>
>
```

Notez la contrainte NOT NULL :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                    primary key,
                    name varchar(255),
                    parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Alternativement, si vous insistez absolument pour que cette association soit unidirectionnelle, vous pouvez déclarer la contrainte NOT NULL sur le mappage <key> :

```
<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children">
      <key column="parent_id" not-null="true"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
```

```

    </class>

</hibernate-mapping>
>

```

D'autre part, si un enfant peut avoir plusieurs parents, une association plusieurs-à-plusieurs est plus appropriée :

```

<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" table="childset">
      <key column="parent_id"/>
      <many-to-many class="Child" column="child_id"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
>

```

Définitions des tables :

```

create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child

```

For more examples and a complete explanation of a parent/child relationship mapping, see [Chapitre 22, Exemple : père/fils](#) for more information.

Des mappages d'association plus exotiques sont possibles, nous cataloguerons toutes les possibilités dans le prochain chapitre.

Mapper les associations

7.1. Introduction

Mapper les associations correctement, est souvent la tâche la plus difficile. Dans cette section, nous traiterons les cas classiques, un par un, en commençant par les mappages unidirectionnels, puis nous aborderons la question des mappages bidirectionnels. Nous illustrons tous nos exemples avec les classes `Person` et `Address`.

Nous classifions les associations selon qu'elles sont ou non bâties sur une table de jointure supplémentaire et sur la multiplicité.

Autoriser une clé étrangère nulle est considéré comme un mauvais choix dans la construction d'un modèle de données. Nous supposons donc que dans tous les exemples qui vont suivre on aura interdit la valeur nulle pour les clés étrangères. Attention, ceci ne veut pas dire que Hibernate ne supporte pas les clés étrangères pouvant prendre des valeurs nulles, et les exemples qui suivent continueront de fonctionner si vous décidez ne plus imposer la contrainte de non-nullité sur les clés étrangères.

7.2. Associations unidirectionnelles

7.2.1. plusieurs-à-un

Une *association plusieurs-à-un unidirectionnelle* est le type que l'on rencontre le plus souvent dans les associations unidirectionnelles.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.2. Un-à-un

Une association *un-à-un sur une clé étrangère* est presque identique. La seule différence est sur la contrainte d'unicité que l'on impose à cette colonne.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Une association *un-à-un unidirectionnelle sur une clé primaire* utilise un générateur d'identifiant particulier. Remarquez que nous avons inversé le sens de cette association dans cet exemple :

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
</class>

<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property"
>person</param>
    </generator>
  </id>
  <one-to-one name="person" constrained="true"/>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
```



```
create table Address ( personId bigint not null primary key )
```

7.2.3. un-à-plusieurs

Une *association un-à-plusieurs unidirectionnelle sur une clé étrangère* est un cas inhabituel, et n'est pas vraiment recommandée.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses">
    <key column="personId"
      not-null="true"/>
    <one-to-many class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

Nous pensons qu'il est préférable d'utiliser une table de jointure pour ce type d'association.

7.3. Associations unidirectionnelles avec tables de jointure

7.3.1. un-à-plusieurs

Une *association unidirectionnelle un-à-plusieurs avec une table de jointure* est un bien meilleur choix. Remarquez qu'en spécifiant `unique="true"`, on a changé la multiplicité plusieurs-à-plusieurs pour un-à-plusieurs.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
```

```
<key column="personId"/>
<many-to-many column="addressId"
  unique="true"
  class="Address"/>
</set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.3.2. plusieurs-à-un

Une *association plusieurs-à-un unidirectionnelle sur une table de jointure* est assez fréquente quand l'association est optionnelle. Par exemple :

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId" unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.3.3. Un-à-un

Une *association unidirectionnelle un-à-un sur une table de jointure* est extrêmement rare mais envisageable.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId"
      unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"
      unique="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
  unique )
create table Address ( addressId bigint not null primary key )
```

7.3.4. Plusieurs-à-plusieurs

Finalement, nous avons un exemple d' *association unidirectionnelle plusieurs-à-plusieurs*.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
```

```
<generator class="native"/>
</id>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
    (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

7.4. Associations bidirectionnelles

7.4.1. un-à-plusieurs / plusieurs-à-un

Une *association bidirectionnelle plusieurs-à-un* est le type d'association que l'on rencontre le plus fréquemment. L'exemple suivant illustre la façon standard de créer des relations parents/enfants.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

Si vous utilisez une `List`, ou toute autre collection indexée, vous devez paramétrer la colonne `key` de la clé étrangère à `not null`, et laisser Hibernate gérer l'association depuis l'extrémité

collection pour maintenir l'index de chaque élément (rendant l'autre extrémité virtuellement inverse en paramétrant `update="false"` et `insert="false"`) :

```
<class name="Person">
  <id name="id" />
  ...
  <many-to-one name="address"
    column="addressId"
    not-null="true"
    insert="false"
    update="false" />
</class>

<class name="Address">
  <id name="id" />
  ...
  <list name="people">
    <key column="addressId" not-null="true" />
    <list-index column="peopleIdx" />
    <one-to-many class="Person" />
  </list>
</class>
>
```

Il est important de définir `not-null="true"` sur l'élément `<key>` du mapping de la collection si la colonne de clé étrangère sous-jacente est NOT NULL. Ne déclarez pas seulement `not-null="true"` sur un élément imbriqué possible `<column>`, mais sur l'élément `<key>`.

7.4.2. Un-à-un

Une association bidirectionnelle un-à-un sur une clé étrangère est assez fréquente :

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native" />
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true" />
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native" />
  </id>
  <one-to-one name="person"
    property-ref="address" />
</class>
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Une association bidirectionnelle un-à-un sur une clé primaire utilise un générateur particulier d'id :

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <one-to-one name="address"/>
</class>

<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property">
>person</param>
    </generator>
  </id>
  <one-to-one name="person"
    constrained="true"/>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. Associations bidirectionnelles avec tables de jointure

7.5.1. un-à-plusieurs / plusieurs-à-un

Une association bidirectionnelle un-à-plusieurs sur une table de jointure. Remarquez que `inverse="true"` peut s'appliquer sur les deux extrémités de l'association, sur la collection, ou sur la jointure.

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses"
    table="PersonAddress">
    <key column="personId"/>
```

```

        <many-to-many column="addressId"
            unique="true"
            class="Address" />
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native" />
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId" />
        <many-to-one name="person"
            column="personId"
            not-null="true" />
    </join>
</class>
>

```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )

```

7.5.2. un-à-un

Une association *bidirectionnelle un-à-un* sur une table de jointure est extrêmement rare mais envisageable.

```

<class name="Person">
    <id name="id" column="personId">
        <generator class="native" />
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true" />
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true" />
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native" />
    </id>
    <join table="PersonAddress"
        optional="true"

```

```
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
    unique )
create table Address ( addressId bigint not null primary key )
```

7.5.3. Plusieurs-à-plusieurs

Finalement nous avons *l'association bidirectionnelle plusieurs-à-plusieurs*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
    (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```


7.6. Des mappages d'associations plus complexes

Des associations encore plus complexes sont *extrêmement* rares. Hibernate permet de gérer des situations plus complexes en utilisant des extraits SQL embarqués dans le fichier de mapping. Par exemple, si une table avec des informations historiques sur un compte définit les colonnes `accountNumber`, `effectiveEndDate` et `effectiveStartDate`, elle sera mappée de la façon suivante :

```
<properties name="currentAccountKey">
  <property name="accountNumber" type="string" not-null="true"/>
  <property name="currentAccount" type="boolean">
    <formula
>case when effectiveEndDate is null then 1 else 0 end</formula>
  </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

Nous pouvons mapper une association à l'instance *courante*, celle avec une `effectiveEndDate` nulle, en utilisant :

```
<many-to-one name="currentAccountInfo"
  property-ref="currentAccountKey"
  class="AccountInfo">
  <column name="accountNumber"/>
  <formula
>'1'</formula>
</many-to-one
>
```

Dans un exemple plus complexe, imaginez qu'une association entre `Employee` et `Organization` soit gérée dans une table `Employment` pleine de données historiques. Dans ce cas, une association vers l'employeur *le plus récent* (celui avec la `startDate` (date de commencement de travail la plus récente) pourrait être mappée comme suit :

```
<join>
  <key column="employeeId"/>
  <subselect>
    select employeeId, orgId
    from Employments
    group by orgId
    having startDate = max(startDate)
  </subselect>
  <many-to-one name="mostRecentEmployer"
    class="Organization"
```

```
        column="orgId"/>  
</join  
>
```

Vous pouvez être créatif grâce à ces possibilités, mais il est généralement plus pratique de gérer ce genre de cas en utilisant des requêtes HQL ou par critère.

Mappage de composants

La notion de *composants* est réutilisée dans différents contextes, avec différents objectifs, à travers Hibernate.

8.1. Objets dépendants

Le composant est un objet inclus dans un autre objet, sauvegardé en tant que type valeur, et non en tant que référence entité. Le terme "composant" fait référence à la notion (au sens objet) de composition et non pas de composant au sens d'architecture de composants. Par exemple, on pourrait modéliser l'objet personne de la façon suivante :

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    .....
    .....
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
}
```

```
void setLast(String last) {
    this.last = last;
}
public char getInitial() {
    return initial;
}
void setInitial(char initial) {
    this.initial = initial;
}
}
```

Maintenant `Name` pourra être sauvegardé en tant que composant de `Person`. Remarquez que `Name` définit des méthodes getter et setter pour ses propriétés persistantes, mais ne doit déclarer aucune interface ou propriété d'identification.

Dans Hibernate le mappage du composant serait :

```
<class name="eg.Person" table="person">
  <id name="Key" column="pid" type="string">
    <generator class="uuid"/>
  </id>
  <property name="birthday" type="date"/>
  <component name="Name" class="eg.Name"
> <!-- class attribute optional -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </component>
</class>
>
```

La table "person" aurait les colonnes `pid`, `birthday`, `initial`, `first` et `last`.

Comme tous les types valeurs, les composants ne supportent pas les références partagées. En d'autres termes, deux instances de `person` peuvent avoir un même nom, mais ces noms sont indépendants, ils peuvent être identiques si on les compare par valeur mais ils représentent deux objets distincts en mémoire. La sémantique de la valeur null d'un composant est *ad hoc*. Quand il recharge l'objet qui contient le composant, Hibernate suppose que si toutes les colonnes de composants sont nulles, le composant est positionné à la valeur null. Ce choix programmatif devrait être satisfaisant dans la plupart des cas.

Les propriétés d'un composant peuvent être de tous les types habituellement supportés par Hibernate (collections, associations plusieurs-à-un, autres composants, etc). Les composants imbriqués ne doivent *pas* être vus comme quelque chose d'exotique. Hibernate a été conçu pour supporter un modèle d'objet finement granulé.

L'élément `<component>` permet de déclarer un sous-élément `<parent>` qui associe une propriété de la classe composant comme une référence arrière vers l'entité contenante.

```
<class name="eg.Person" table="person">
```

```

<id name="Key" column="pid" type="string">
  <generator class="uuid"/>
</id>
<property name="birthday" type="date"/>
<component name="Name" class="eg.Name" unique="true">
  <parent name="namedPerson"/> <!-- reference back to the Person -->
  <property name="initial"/>
  <property name="first"/>
  <property name="last"/>
</component>
</class>
>

```

8.2. Collection d'objets dépendants

Les collections d'objets dépendants sont supportées (exemple: un tableau de type `Name`). Déclarez votre collection de composants en remplaçant la balise `<element>` par la balise `<composite-element>` :

```

<set name="someNames" table="some_names" lazy="true">
  <key column="id"/>
  <composite-element class="eg.Name"
> <!-- class attribute required -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </composite-element>
</set>
>

```



Important

Remarque : si vous définissez un `Set` d'éléments composites, il est très important d'implémenter les méthodes `equals()` et `hashCode()` correctement.

Les éléments composites peuvent aussi contenir des composants mais pas des collections. Si votre élément composite contient aussi des composants, utilisez la balise `<nested-composite-element>`. Une collection de composants qui contiennent eux-mêmes des composants est un cas très exotique. A ce stade, demandez-vous si une association un-à-plusieurs ne serait pas plus appropriée. Essayez de remodeler votre élément composite comme une entité - remarquez que si le modèle Java est le même, toutefois le modèle relationnel et la sémantique de persistance diffèrent quelque peu.

Remarquez que le mappage d'éléments composites ne supporte pas la nullité des propriétés lorsqu'on utilise un `<set>`. Hibernate lorsqu'il supprime un objet, utilise chaque colonne pour identifier un objet (il n'y a pas de colonne distincte de clés primaires dans la table d'éléments composites), ce qui n'est pas possible avec des valeurs nulles. Vous devez donc choisir d'interdire

la nullité des propriétés d'un élément composite ou choisir un autre type de collection comme : `<list>`, `<map>`, `<bag>` OU `<idbag>`.

Un cas particulier d'élément composite est un élément composite qui inclut un élément imbriqué `<many-to-one>`. Un mappage comme celui-ci vous permet d'associer des colonnes supplémentaires d'une table d'association plusieurs à plusieurs à la classe de l'élément composite. L'exemple suivant est une association plusieurs à plusieurs de `Order` à `Item` où `purchaseDate`, `price` et `quantity` sont des propriétés de l'association :

```
<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items" lazy="true">
    <key column="order_id">
      <composite-element class="eg.Purchase">
        <property name="purchaseDate"/>
        <property name="price"/>
        <property name="quantity"/>
        <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
      </composite-element>
    </set>
  </class>
>
```

Par ailleurs, on ne peut évidemment pas faire référence à l'achat (`purchase`), pour pouvoir naviguer de façon bidirectionnelle dans l'association. N'oubliez pas que les composants sont de type valeurs et n'autorisent pas les références partagées. Un `Purchase` unique peut être dans le set d'un `Order`, mais ne peut pas être référencé par `Item` simultanément.

Même les associations ternaires, quaternaires ou autres sont possibles :

```
<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items" lazy="true">
    <key column="order_id">
      <composite-element class="eg.OrderLine">
        <many-to-one name="purchaseDetails" class="eg.Purchase"/>
        <many-to-one name="item" class="eg.Item"/>
      </composite-element>
    </set>
  </class>
>
```

Des éléments composites peuvent apparaître dans les requêtes en utilisant la même syntaxe que les associations vers d'autres entités.

8.3. Les composants en tant qu'indices de Map

L'élément `<composite-map-key>` vous permet de mapper une classe de composant comme indice d'une `Map`. Assurez-vous de surcharger correctement `hashCode()` et `equals()` dans la classe du composant.

8.4. Les composants en tant qu'identifiants composites

Vous pouvez utiliser un composant comme identifiant d'une classe entité. À cet effet, votre classe de composant doit respecter certaines exigences :

- Elle doit implémenter `java.io.Serializable`.
- Elle doit redéfinir `equals()` et `hashCode()`, de façon cohérente avec la notion d'égalité de clé composite de la base de données.



Remarque

Avec Hibernate3, la seconde exigence n'est plus absolument nécessaire, néanmoins continuez de l'effectuer.

Vous ne pouvez pas utiliser de `IdentifierGenerator` pour générer des clés composites, par contre l'application doit assigner ses propres identifiants.

Utiliser la balise `<composite-id>` (avec les éléments imbriqués `<key-property>`) à la place de l'habituel déclaration `<id>`. Par exemple, la classe `OrderLine` possède une clé primaire qui dépend de la clé primaire (composite) de `Order`.

```
<class name="OrderLine">

  <composite-id name="id" class="OrderLineId">
    <key-property name="lineId"/>
    <key-property name="orderId"/>
    <key-property name="customerId"/>
  </composite-id>

  <property name="name"/>

  <many-to-one name="order" class="Order"
    insert="false" update="false">
    <column name="orderId"/>
    <column name="customerId"/>
  </many-to-one>
  ...
</class>
>
```

Toutes les clés étrangères référençant la table `OrderLine` sont également composites. Vous devez en tenir compte lorsque vous écrivez vos mappage d'association pour les autres classes. Une association à `OrderLine` sera mappée de la façon suivante :

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
  <column name="lineId"/>
  <column name="orderId"/>
  <column name="customerId"/>
</many-to-one>
>
```



Astuce

The `column` element is an alternative to the `column` attribute everywhere. Using the `column` element just gives more declaration options, which are mostly useful when utilizing `hbm2ddl`.

Une association plusieurs-à-plusieurs à `OrderLine` utilisera aussi une clé étrangère composite :

```
<set name="undeliveredOrderLines">
  <key column name="warehouseId"/>
  <many-to-many class="OrderLine">
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
  </many-to-many>
</set>
>
```

La collection des `OrderLine` s dans `Order` utilisera :

```
<set name="orderLines" inverse="true">
  <key>
    <column name="orderId"/>
    <column name="customerId"/>
  </key>
  <one-to-many class="OrderLine"/>
</set>
>
```

Comme d'habitude, l'élément `<one-to-many>` ne déclare pas de colonne.

Si `OrderLine` lui-même possède une collection, il possédera de même une clé composite étrangère.


```

<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key
>      <!-- a collection inherits the composite key type -->
          <column name="lineId"/>
          <column name="orderId"/>
          <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
>

```

8.5. Les composants dynamiques

Vous pouvez également mapper une propriété de type `Map` :

```

<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
>

```

La sémantique de l'association à un `<dynamic-component>` est identique à celle que l'on utilise pour le `<component>`. L'avantage de ce type de mappage est qu'il permet de déterminer les véritables propriétés du bean au moment du déploiement, en éditant simplement le document de mappage. La manipulation du document de mappage pendant l'exécution de l'application est aussi possible en utilisant un parser DOM. Il y a même mieux, vous pouvez accéder (et changer) le métamodèle de configuration-temps de Hibernate en utilisant l'objet `Configuration`.

Mapping d'héritage de classe

9.1. Les trois stratégies

Hibernate supporte les trois stratégies d'héritage de base :

- une table par hiérarchie de classe
- table per subclass
- une table par classe concrète

Par ailleurs, Hibernate supporte une quatrième stratégie, avec un polymorphisme légèrement différent :

- le polymorphisme implicite

Il est possible d'utiliser différentes stratégies de mapping pour différentes branches d'une même hiérarchie d'héritage, et ensuite d'employer le polymorphisme implicite pour réaliser le polymorphisme à travers toute la hiérarchie. Toutefois, Hibernate ne supporte pas les mélanges de mappages `<subclass>`, `<joined-subclass>` et `<union-subclass>` pour le même élément `<class>` racine. Il est possible de mélanger les stratégies d'une table par hiérarchie et d'une table par sous-classe, pour le même élément `<class>`, en combinant les éléments `<subclass>` et `<join>` (voir ci-dessous).

Il est possible de définir des mappages de `subclass`, `union-subclass`, et `joined-subclass` dans des documents de mappage séparés, directement sous `hibernate-mapping`. Ceci vous permet d'étendre une hiérarchie de classe juste en ajoutant un nouveau fichier de mappage. Vous devez spécifier un attribut `extends` dans le mappage de la sous-classe, en nommant une super-classe précédemment mappée. Note : précédemment cette fonctionnalité rendait important l'ordre des documents de mappage. Depuis Hibernate3, l'ordre des fichiers de mappage n'importe plus lors de l'utilisation du mot-clé "extends". L'ordre à l'intérieur d'un simple fichier de mappage impose encore de définir les classes mères avant les classes filles.

```
<hibernate-mapping>
  <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
    <property name="name" type="string"/>
  </subclass>
</hibernate-mapping>
>
```

9.1.1. Une table par hiérarchie de classe

Supposons que nous ayons une interface `Payment`, implémentée par `CreditCardPayment`, `CashPayment`, `ChequePayment`. La stratégie une table par hiérarchie serait :

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native" />
  </id>
  <discriminator column="PAYMENT_TYPE" type="string"/>
  <property name="amount" column="AMOUNT" />
  ...
  <subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <property name="creditCardType" column="CCTYPE" />
    ...
  </subclass>
  <subclass name="CashPayment" discriminator-value="CASH">
    ...
  </subclass>
  <subclass name="ChequePayment" discriminator-value="CHEQUE">
    ...
  </subclass>
</class>
>
```

Une seule table est requise. Une grande limitation de cette stratégie est que les colonnes déclarées par les classes filles, telles que CCTYPE, peuvent ne pas avoir de contrainte NOT NULL.

9.1.2. Une table par classe fille

Une table par classe-fille de mappage serait :

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native" />
  </id>
  <property name="amount" column="AMOUNT" />
  ...
  <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <key column="PAYMENT_ID" />
    <property name="creditCardType" column="CCTYPE" />
    ...
  </joined-subclass>
  <joined-subclass name="CashPayment" table="CASH_PAYMENT">
    <key column="PAYMENT_ID" />
    ...
  </joined-subclass>
  <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    <key column="PAYMENT_ID" />
    ...
  </joined-subclass>
</class>
>
```

Quatre tables sont requises. Les trois tables des classes filles ont une clé primaire associée à la table classe mère (le modèle relationnel est une association un-à-un).

9.1.3. Une table par classe fille, en utilisant un discriminant

Notez que l'implémentation Hibernate de la stratégie une table par classe fille, ne nécessite pas de colonne discriminante dans la table classe mère. D'autres implémentations de mappers Objet/Relationnel utilisent une autre implémentation de la stratégie une table par classe fille qui nécessite une colonne de type discriminant dans la table de la classe mère. L'approche prise par Hibernate est plus difficile à implémenter mais plus correcte d'un point de vue relationnel. Si vous aimeriez utiliser une colonne discriminante avec la stratégie d'une table par classe fille, vous pouvez combiner l'utilisation de `<subclass>` et `<join>`, comme suit :

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <discriminator column="PAYMENT_TYPE" type="string"/>
  <property name="amount" column="AMOUNT"/>
  ...
  <subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <join table="CREDIT_PAYMENT">
      <key column="PAYMENT_ID"/>
      <property name="creditCardType" column="CCTYPE"/>
      ...
    </join>
  </subclass>
  <subclass name="CashPayment" discriminator-value="CASH">
    <join table="CASH_PAYMENT">
      <key column="PAYMENT_ID"/>
      ...
    </join>
  </subclass>
  <subclass name="ChequePayment" discriminator-value="CHEQUE">
    <join table="CHEQUE_PAYMENT" fetch="select">
      <key column="PAYMENT_ID"/>
      ...
    </join>
  </subclass>
</class>
>
```

La déclaration optionnelle `fetch="select"` indique à Hibernate de ne pas récupérer les données de la classe fille `ChequePayment` par une jointure externe lors des requêtes sur la classe mère.

9.1.4. Mélange d'une table par hiérarchie de classe avec une table par classe fille

Vous pouvez même mélanger les stratégies d'une table par hiérarchie de classe et d'une table par classe fille en utilisant cette approche :

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
```

```
<generator class="native"/>
</id>
<discriminator column="PAYMENT_TYPE" type="string"/>
<property name="amount" column="AMOUNT"/>
...
<subclass name="CreditCardPayment" discriminator-value="CREDIT">
  <join table="CREDIT_PAYMENT">
    <property name="creditCardType" column="CCTYPE"/>
    ...
  </join>
</subclass>
<subclass name="CashPayment" discriminator-value="CASH">
  ...
</subclass>
<subclass name="ChequePayment" discriminator-value="CHEQUE">
  ...
</subclass>
</class>
>
```

Pour importe laquelle de ces stratégies, une association polymorphique vers la classe racine `Payment` est mappée en utilisant `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

9.1.5. Une table par classe concrète

Il y a deux manières d'utiliser la stratégie d'une table par classe concrète. La première est d'employer `<union-subclass>`.

```
<class name="Payment">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="sequence"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <property name="creditCardType" column="CCTYPE"/>
    ...
  </union-subclass>
  <union-subclass name="CashPayment" table="CASH_PAYMENT">
    ...
  </union-subclass>
  <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    ...
  </union-subclass>
</class>
>
```

Trois tables sont nécessaires pour les classes filles. Chaque table définit des colonnes pour toutes les propriétés de la classe, y compris les propriétés héritées.

La limitation de cette approche est que si une propriété est mappée sur la classe mère, le nom de la colonne doit être le même pour toutes les classes filles (Une future version de Hibernate pourra assouplir ce comportement). La stratégie du générateur d'identifiant n'est pas permise dans l'héritage de classes filles par union, en effet la valeur de graine de la clef primaire doit être partagée par toutes les classes filles fusionnées d'une hiérarchie.

Si votre classe mère est abstraite, mappez la avec `abstract="true"`. Bien sûr, si elle n'est pas abstraite, une table supplémentaire (par défaut, `PAYMENT` dans l'exemple ci-dessus) est requise pour contenir des instances de la classe mère.

9.1.6. Une table par classe concrète, en utilisant le polymorphisme implicite

Une approche alternative est l'emploi du polymorphisme implicite :

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
  <id name="id" type="long" column="CREDIT_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="CREDIT_AMOUNT"/>
  ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
  <id name="id" type="long" column="CASH_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="CASH_AMOUNT"/>
  ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
  <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="CHEQUE_AMOUNT"/>
  ...
</class>
>
```

Notice that the `Payment` interface is not mentioned explicitly. Also notice that properties of `Payment` are mapped in each of the subclasses. If you want to avoid duplication, consider using XML entities (for example, [`<!ENTITY allproperties SYSTEM "allproperties.xml">]` in the DOCTYPE declaration and `&allproperties;` in the mapping).

L'inconvénient de cette approche est que Hibernate ne génère pas de SQL `UNION` s lors de l'exécution des requêtes polymorphiques.

Pour cette stratégie de mappage, une association polymorphique pour `Payment` est habituellement mappée en utilisant `<any>`.

```
<any name="payment" meta-type="string" id-type="long">
  <meta-value value="CREDIT" class="CreditCardPayment" />
  <meta-value value="CASH" class="CashPayment" />
  <meta-value value="CHEQUE" class="ChequePayment" />
  <column name="PAYMENT_CLASS" />
  <column name="PAYMENT_ID" />
</any>
>
```

9.1.7. Mélange du polymorphisme implicite avec d'autres mappages d'héritage

Il y a une chose supplémentaire à noter à propos de ce mappage. Puisque les classes filles sont chacune mappées avec leur propre élément `<class>` (et puisque `Payment` est juste une interface), chaque classe fille pourrait facilement faire partie d'une autre hiérarchie d'héritage ! (Et vous pouvez encore faire des requêtes polymorphiques pour l'interface `Payment`).

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
  <id name="id" type="long" column="CREDIT_PAYMENT_ID">
    <generator class="native" />
  </id>
  <discriminator column="CREDIT_CARD" type="string" />
  <property name="amount" column="CREDIT_AMOUNT" />
  ...
  <subclass name="MasterCardPayment" discriminator-value="MDC" />
  <subclass name="VisaPayment" discriminator-value="VISA" />
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
  <id name="id" type="long" column="TXN_ID">
    <generator class="native" />
  </id>
  ...
  <joined-subclass name="CashPayment" table="CASH_PAYMENT">
    <key column="PAYMENT_ID" />
    <property name="amount" column="CASH_AMOUNT" />
    ...
  </joined-subclass>
  <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    <key column="PAYMENT_ID" />
    <property name="amount" column="CHEQUE_AMOUNT" />
    ...
  </joined-subclass>
</class>
>
```

Encore une fois, nous ne mentionnons pas explicitement `Payment`. Si nous exécutons une requête sur l'interface `Payment` - par exemple, `from Payment` - Hibernate retournera automatiquement les instances de `CreditCardPayment` (et ses classes filles puisqu'elles implémentent aussi `Payment`), `CashPayment` et `ChequePayment` mais pas les instances de `NonelectronicTransaction`.

9.2. Limitations

Il y a certaines limitations à l'approche du "polymorphisme implicite" pour la stratégie de mappage d'une table par classe concrète. Il y a plutôt moins de limitations restrictives aux mappages `<union-subclass>`.

The following table shows the limitations of table per concrete-class mappings, and of implicit polymorphism, in Hibernate.

Tableau 9.1. Features of inheritance mappings

Inheritance strategy	Polymorphic many-to-one	Polymorphic one-to-one	Polymorphic one-to-many	Polymorphic many-to-many	Polymorphic load()/get()	Polymorphic queries	Polymorphic joins	Outer join fetching
table per class-hierarchy	<code><many-to-one></code>	<code><one-to-one></code>	<code><one-to-many></code>	<code><many-to-many></code>	<code>s.get(Payment.class, id)</code>	<code>from Payment p</code>	<code>Order o join o.payment p</code>	<i>supported</i>
table per subclass	<code><many-to-one></code>	<code><one-to-one></code>	<code><one-to-many></code>	<code><many-to-many></code>	<code>s.get(Payment.class, id)</code>	<code>from Payment p</code>	<code>Order o join o.payment p</code>	<i>supported</i>
une table par classe concrète (union-classe fille)	<code><many-to-one></code>	<code><one-to-one></code>	<code><one-to-many></code> (for <code>inverse="true"</code> only)	<code><many-to-many></code>	<code>s.get(Payment.class, id)</code>	<code>from Payment p</code>	<code>Order o join o.payment p</code>	<i>supported</i>
table per concrete class (implicit polymorphism)	<code><any></code>	<i>not supported</i>	<i>not supported</i>	<code><many-to-many></code>	<code>s.createCriteria(Payment.class).add(Restrictions.</code>	<code>from Payment p</code>	<i>not supported</i>	<i>not supported</i>

Travailler avec des objets

Hibernate est une solution de mappage objet/relationnel complète qui ne masque pas seulement au développeur les détails du système de gestion de base de données sous-jacent, mais offre aussi *la gestion d'état* des objets. C'est, contrairement à la gestion de `statements` SQL dans les couches de persistance habituelles JDBC/SQL, une vue orientée objet très naturelle de la persistance dans les applications Java.

En d'autres termes, les développeurs d'applications Hibernate devraient toujours réfléchir à *l'état* de leurs objets, et pas nécessairement à l'exécution des expressions SQL. Cette part est prise en charge par Hibernate et importante seulement aux développeurs d'applications lors du réglage de la performance de leur système.

10.1. États des objets Hibernate

Hibernate définit et prend en charge les états d'objets suivants :

- *Éphémère* (transient) - un objet est éphémère s'il a juste été instancié en utilisant l'opérateur `new`. Il n'a aucune représentation persistante dans la base de données et aucune valeur d'identifiant n'a été assignée. Les instances éphémères seront détruites par le ramasse-miettes si l'application n'en conserve aucune référence. Utilisez la `Session` d'Hibernate pour rendre un objet persistant (et laisser Hibernate s'occuper des expressions SQL qui ont besoin d'être exécutées pour cette transistion).
- *Persistant* - une instance persistante a une représentation dans la base de données et une valeur d'identifiant. Elle pourrait avoir juste été sauvegardée ou chargée, pourtant, elle est par définition dans la portée d'une `Session`. Hibernate détectera tout changement effectué sur un objet dans l'état persistant et synchronisera l'état avec la base de données lors de la fin de l'unité de travail. Les développeurs n'exécutent pas d'expressions `UPDATE` ou `DELETE` manuelles lorsqu'un objet devrait être rendu éphémère.
- *Détaché* - une instance détachée est un objet qui a été persistant, mais dont la `Session` a été fermée. La référence à l'objet est encore valide, bien sûr, et l'instance détachée pourrait même être modifiée dans cet état. Une instance détachée peut être rattachée à une nouvelle `Session` ultérieurement, la rendant (et toutes les modifications avec) de nouveau persistante. Cette fonctionnalité rend possible un modèle de programmation pour de longues unités de travail qui requièrent un temps de réflexion de l'utilisateur. Nous les appelons des *conversations*, c'est-à-dire une unité de travail du point de vue de l'utilisateur.

Nous allons maintenant approfondir le sujet des états et des transitions d'état (et des méthodes Hibernate qui déclenchent une transition).

10.2. Rendre des objets persistants

Newly instantiated instances of a persistent class are considered *transient* by Hibernate. We can make a transient instance *persistent* by associating it with a session:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

Si `Cat` a un identifiant généré, l'identifiant est généré et assigné au `cat` lorsque `save()` est appelé. Si `Cat` a un identifiant `assigned`, ou une clef composée, l'identifiant devrait être assigné à l'instance de `cat` avant d'appeler `save()`. Vous pouvez aussi utiliser `persist()` à la place de `save()`, avec la sémantique définie plus tôt dans la première ébauche d'EJB3.

- `persist()` rend une instance éphémère persistante. Toutefois, il ne garantit pas que la valeur d'identificateur soit affectée à l'instance permanente immédiatement, l'affectation peut se produire au moment de `flush`. `Persist()` garantit également qu'il ne s'exécutera pas un énoncé `INSERT` s'il est appelée en dehors des limites de transaction. C'est utile pour les longues conversations dans un contexte de session/persistance étendu.
- `save()` garantit le retour d'un identifiant. Si une instruction `INSERT` doit être exécutée pour obtenir l'identifiant (par exemple, le générateur "identity", et non pas "sequence"), cet `INSERT` se produit immédiatement, que vous soyez à l'intérieur ou à l'extérieur d'une transaction. C'est problématique dans une conversation longue dans un contexte de session/persistance étendu.

Alternativement, vous pouvez assigner l'identifiant en utilisant une version surchargée de `save()`.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

Si l'objet que vous rendez persistant a des objets associés (par exemple, la collection `kittens` dans l'exemple précédent), ces objets peuvent être rendus persistants dans n'importe quel ordre désiré, à moins que vous ayez une contrainte `NOT NULL` sur la colonne de la clé étrangère. Il n'y a jamais de risque de violer une contrainte de cl. étrangère. Cependant, vous pourriez violer une contrainte `NOT NULL` si vous appeliez `save()` sur les objets dans le mauvais ordre.

Habituellement, vous ne vous préoccupez pas de ce détail, puisque vous utiliserez très probablement la fonctionnalité de *persistance transitive* de Hibernate pour sauvegarder les objets associés automatiquement. Alors, même les violations de contrainte `NOT NULL` n'ont plus lieu - Hibernate prendra soin de tout. La persistance transitive est traitée plus loin dans ce chapitre.

10.3. Chargement d'un objet

Les méthodes `load()` de `Session` vous donnent un moyen de récupérer une instance persistante si vous connaissez déjà son identifiant. `load()` prend un objet de classe et chargera l'état dans une instance nouvellement instanciée de cette classe, dans un état persistant.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

Alternativement, vous pouvez charger un état dans une instance donnée :

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Notez que `load()` lèvera une exception irrécupérable s'il n'y a pas de ligne correspondante dans la base de données. Si la classe est mappée avec un proxy, `load()` retourne juste un proxy non initialisé et n'accède en fait pas à la base de données jusqu'à ce que vous invoquiez une méthode du proxy. Ce comportement est très utile si vous souhaitez créer une association vers un objet sans réellement le charger à partir de la base de données. Cela permet aussi à de multiples instances d'être chargées comme un lot si `batch-size` est défini pour le mapping de la classe.

Si vous n'êtes pas certain qu'une ligne correspondante existe, vous utiliserez la méthode `get()`, laquelle accède à la base de données immédiatement et retourne `null` s'il n'y a pas de ligne correspondante.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Vous pouvez même charger un objet en employant un SQL `SELECT ... FOR UPDATE`, en utilisant un `LockMode`. Voir la documentation de l'API pour plus d'informations.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Notez que n'importe quelle instance associée ou collection contenue *ne sont pas* sélectionnées par `FOR UPDATE`, à moins que vous ne décidiez de spécifier `lock` ou `all` en tant que style de cascade pour l'association.

Il est possible de re-charger un objet et toutes ses collections à tout moment, en utilisant la méthode `refresh()`. C'est utile lorsque des "triggers" de base de données sont utilisés pour initialiser certaines propriétés de l'objet.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

How much does Hibernate load from the database and how many SQL `SELECTS` will it use? This depends on the *fetching strategy*. This is explained in [Section 20.1, « Stratégies de chargement »](#).

10.4. Requêtage

Si vous ne connaissez pas les identifiants des objets que vous recherchez, vous avez besoin d'une requête. Hibernate supporte un langage de requêtes orientées objet, facile à utiliser mais puissant. Pour la création de requêtes par programmation, Hibernate supporte une fonction de requêtage sophistiquée Criteria et Example (QBC et QBE). Vous pouvez aussi exprimer votre requête dans le SQL natif de votre base de données, avec un support optionnel de Hibernate pour la conversion des ensembles de résultats en objets.

10.4.1. Exécution de requêtes

Les requêtes HQL et SQL natives sont représentées avec une instance de `org.hibernate.Query`. L'interface offre des méthodes pour la liaison des paramètres, la gestion des ensembles de résultats, et pour l'exécution de la requête réelle. Vous obtenez toujours une `Query` en utilisant la Session courante :

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
```

```

        .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());

```

Une requête est généralement exécutée en invoquant `list()`, le résultat de la requête sera chargée complètement dans une collection en mémoire. Les instances d'entités récupérées par une requête sont dans un état persistant. La méthode `uniqueResult()` offre un raccourci si vous savez que votre requête retournera un seul objet. Notez que les requêtes qui utilisent le chargement agressif de collections retournent habituellement des copies des objets racine (mais avec leurs collections initialisées). Vous pouvez simplement filtrer ces copies via un `Set`.

10.4.1.1. Itération de résultats

Parfois, vous serez en mesure d'obtenir de meilleures performances en exécutant la requête avec la méthode `iterate()`. En général, ce sera uniquement le cas si vous attendez que les instances réelles d'entité retournées par la requête, soient déjà chargées dans la session ou le cache de second niveau. Si elles ne sont pas déjà cachées, `iterate()` sera plus lent que `list()` et pourrait nécessiter plusieurs accès à la base de données pour une simple requête, généralement 1 pour le select initial qui retourne seulement les identifiants, et n selects supplémentaires pour initialiser les instances réelles.

```

// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next(); // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}

```

10.4.1.2. Requetes qui retournent des tuples

Les requêtes d'Hibernate retournent parfois des tuples d'objets, auquel cas chaque tuple est retourné comme un tableau :

```

Iterator kittensAndMothers = sess.createQuery(
    "select kitten, mother from Cat kitten join kitten.mother mother")
    .list()
    .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
}

```

```
Cat mother = (Cat) tuple[1];
....
}
```

10.4.1.3. Résultats scalaires

Certaines requêtes peuvent spécifier une propriété de classe dans la clause `select`. Elles peuvent même appeler des fonctions d'agrégat SQL. Les propriétés ou les agrégats sont considérés comme des résultats "scalaires" (et non des entités dans un état persistant).

```
Iterator results = sess.createQuery(
    "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
    "group by cat.color")
    .list()
    .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    ....
}
```

10.4.1.4. Lier des paramètres

Des méthodes de `Query` sont fournies pour lier des valeurs à des paramètres nommés ou à des paramètres de style JDBC `?`. *Contrairement à JDBC, les numéros des paramètres de Hibernate commencent à zéro.* Les paramètres nommés sont des identifiants de la forme `:nom` dans la chaîne de caractères de la requête. Les avantages des paramètres nommés sont :

- les paramètres nommés sont insensibles à l'ordre dans lequel ils apparaissent dans la chaîne de la requête
- ils peuvent apparaître plusieurs fois dans la même requête
- ils sont auto-documentés

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```



```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

10.4.1.5. Pagination

Si vous avez besoin de spécifier des liens sur votre ensemble de résultats (le nombre maximum de lignes et/ou la première ligne que vous voulez récupérer) vous utiliserez des méthodes de l'interface `Query` :

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate sait comment traduire cette requête de limite en SQL natif pour votre SGBD.

10.4.1.6. Itération "scrollable"

Si votre connecteur JDBC supporte les `ResultSet` s "scrollables", l'interface `Query` peut être utilisée pour obtenir un objet `ScrollableResults`, qui permettra une navigation flexible dans les résultats de la requête.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE
> i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close();
```

Notez qu'une connexion ouverte (et un curseur) est requise pour cette fonctionnalité, utilisez `setMaxResult()/setFirstResult()` si vous avez besoin d'une fonctionnalité de pagination hors ligne.

10.4.1.7. Externaliser des requêtes nommées

Vous pouvez aussi définir des requêtes nommées dans le document de mapping. Souvenez-vous d'utiliser une section `CDATA` si votre requête contient des caractères qui pourraient être interprétés comme des éléments XML.

```
<query name="ByNameAndMaximumWeight"
><![CDATA[
    from eg.DomesticCat as cat
      where cat.name = ?
      and cat.weight
> ?
] ]></query
>
```

La liaison de paramètres et l'exécution sont effectués par programmation :

```
Query q = sess.getNamedQuery( "ByNameAndMaximumWeight" );
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Notez que le code réel du programme est indépendant du langage de requête utilisé, vous pouvez aussi définir des requêtes SQL natives dans les méta-données, ou migrer des requêtes existantes vers Hibernate en les plaçant dans les fichiers de mapping.

Notez aussi que la déclaration d'une requête dans un élément `<hibernate-mapping>` nécessite un nom globalement unique pour la requête, alors que la déclaration d'une requête dans un élément `<class>` est rendue unique de manière automatique par la mise en préfixe du nom entièrement qualifié de la classe, par exemple `eg.Cat.ByNameAndMaximumWeight`.

10.4.2. Filtrer des collections

Un *filtre* de collection est un type spécial de requête qui peut être appliqué à une collection persistante ou à un tableau. La chaîne de requêtes peut se référer à `this`, correspondant à l'élément de la collection courant.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?"
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
```

```
);
```

La collection retournée est considérée comme un bag, et c'est une copie de la collection donnée. La collection originale n'est pas modifiée. C'est contraire à l'implication du nom "filtre"; mais cohérent avec le comportement attendu.

Observez que les filtres ne nécessitent pas une clause `from` (bien qu'ils puissent en avoir une si besoin est). Les filtres ne sont pas limités à retourner des éléments de la collection eux-mêmes.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue"
).list();
```

Même une requête de filtre vide est utile, par exemple pour charger un sous-ensemble d'éléments dans une énorme collection :

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), ""
).setFirstResult(0).setMaxResults(10)
.list();
```

10.4.3. Requêtes par critères

HQL est extrêmement puissant, mais certains développeurs préfèrent construire des requêtes dynamiquement, en utilisant l'API orientée objet, plutôt que de construire des chaînes de requêtes. Hibernate fournit une API intuitive de requête `Criteria` pour ces cas :

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The `Criteria` and the associated `Example` API are discussed in more detail in [Chapitre 16, Requêtes par critères](#).

10.4.4. Requêtes en SQL natif

Vous pouvez exprimer une requête en SQL, en utilisant `createSQLQuery()` et laisser Hibernate s'occuper du mappage des résultats vers des objets. Notez que vous pouvez à tout moment, appeler `session.connection()` et utiliser directement la `Connection JDBC`. Si vous choisissez d'utiliser l'API Hibernate, vous devez mettre les alias SQL entre accolades :

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
```

```
.addEntity("cat", Cat.class)
.list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
        "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
    .list();
```

SQL queries can contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in [Chapitre 17, SQL natif](#).

10.5. Modifier des objets persistants

Les *instances persistantes transactionnelles* (c'est-à-dire des objets chargés, sauvegardés, créés ou requêtés par la *Session*) peuvent être manipulés par l'application et tout changement vers l'état persistant sera persisté lorsque la *Session* est "*flushée*" (traité plus tard dans ce chapitre). Il n'est pas nécessaire d'appeler une méthode particulière (comme `update()`, qui a un but différent) pour rendre vos modifications persistantes. Donc la manière la plus directe de mettre à jour l'état d'un objet est de le charger avec `load()`, et puis de le manipuler directement, tant que la *Session* est ouverte :

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush(); // changes to cat are automatically detected and persisted
```

Parfois ce modèle de programmation est inefficace puisqu'il nécessiterait un SQL `SELECT` (pour charger l'objet) et un SQL `UPDATE` (pour persister son état mis à jour) dans la même session. Ainsi Hibernate offre une autre approche, en utilisant des instances détachées.



Important

Hibernate does not offer its own API for direct execution of `UPDATE` or `DELETE` statements. Hibernate is a *state management* service, you do not have to think in *statements* to use it. JDBC is a perfect API for executing SQL statements, you can get a JDBC *Connection* at any time by calling `session.connection()`. Furthermore, the notion of mass operations conflicts with object/relational mapping for online transaction processing-oriented applications. Future versions of Hibernate can, however, provide special mass operation functions. See [Chapitre 14, Traitement par lot](#) for some possible batch operation tricks.

10.6. Modifier des objets détachés

Beaucoup d'applications ont besoin de récupérer un objet dans une transaction, de l'envoyer à la couche interfacée avec l'utilisateur pour les manipulations, et de sauvegarder les changements dans une nouvelle transaction. Les applications qui utilisent cette approche dans un environnement à haute concurrence utilisent généralement des données versionnées pour assurer l'isolation des "longues" unités de travail.

Hibernate supporte ce modèle en permettant pour le rattachement d'instances détachées en utilisant des méthodes `Session.update()` ou `Session.merge()` :

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat); // update cat
secondSession.update(mate); // update mate
```

Si le `Cat` avec l'identifiant `catId` avait déjà été chargé par `secondSession` lorsque l'application a essayé de le rattacher, une exception aurait été levée.

Utilisez `update()` si vous êtes sûr que la session ne contient pas déjà une instance persistante avec le même identifiant, et `merge()` si vous voulez fusionner vos modifications n'importe quand sans considérer l'état de la session. En d'autres termes, `update()` est généralement la première méthode que vous devez appeler dans une session fraîche, pour vous assurer que le rattachement de vos instances détachées est la première opération qui est exécutée.

The application should individually `update()` detached instances that are reachable from the given detached instance *only* if it wants their state to be updated. This can be automated using *transitive persistence*. See [Section 10.11, « Persistence transitive »](#) for more information.

La méthode `lock()` permet aussi à une application de ré-associer un objet avec une nouvelle session. Cependant, l'instance détachée doit être non modifiée.

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

Notez que `lock()` peut être utilisé avec différents `LockMode`s, voir la documentation de l'API et le chapitre sur la gestion des transactions pour plus d'informations. Le rattachement n'est pas le seul cas d'utilisation pour `lock()`.

Other models for long units of work are discussed in [Section 12.3, « Contrôle de concurrence optimiste »](#).

10.7. Détection automatique d'un état

Les utilisateurs d'Hibernate ont demandé une méthode dont l'intention générale serait soit de sauvegarder une instance éphémère en générant un nouvel identifiant, soit mettre à jour/rattacher les instances détachées associées à l'identifiant courant. La méthode `saveOrUpdate()` implémente cette fonctionnalité.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat); // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate); // save the new instance (mate has a null id)
```

L'usage et la sémantique de `saveOrUpdate()` semble être confuse pour les nouveaux utilisateurs. Premièrement, aussi longtemps que vous n'essayez pas d'utiliser des instances d'une session dans une autre, vous ne devriez pas avoir besoin d'utiliser `update()`, `saveOrUpdate()`, ou `merge()`. Certaines applications n'utiliseront jamais ces méthodes.

Généralement `update()` ou `saveOrUpdate()` sont utilisées dans le scénario suivant :

- l'application charge un objet dans la première session
- l'objet est passé à la couche utilisateur
- certaines modifications sont effectuées sur l'objet
- l'objet est retourné à la couche logique métier
- l'application persiste ces modifications en appelant `update()` dans une seconde session

`saveOrUpdate()` s'utilise dans le cas suivant :

- si l'objet est déjà persistant dans cette session, ne rien faire
- si un autre objet associé à la session a le même identifiant, lever une exception
- si l'objet n'a pas de propriété d'identifiant, appeler `save()`
- si l'identifiant de l'objet a une valeur assignée à un objet nouvellement instancié, appeler `save()`
- si l'objet est versionné (par `<version>` ou `<timestamp>`), et la valeur de la propriété de version est la même valeur que celle assignée à un objet nouvellement instancié, appeler `save()`
- sinon mettre à jour l'objet avec `update()`

et `merge()` est très différent :

- s'il y a une instance persistante avec le même identifiant couramment associé à la session, copier l'état de l'objet donné dans l'instance persistante
- s'il n'y a pas d'instance persistante associée à cette session, essayer de le charger à partir de la base de données, ou créer une nouvelle instance persistante
- l'instance persistante est retournée
- l'instance donnée ne devient pas associée à la session, elle reste détachée

10.8. Suppression d'objets persistants

`Session.delete()` supprimera l'état d'un objet de la base de données. Bien sûr, votre application pourrait encore conserver une référence vers un objet effacé. Il est préférable de penser à `delete()` comme rendant une instance persistante éphémère.

```
sess.delete(cat);
```

Vous pouvez effacer des objets dans l'ordre que vous voulez, sans risque de violations de contrainte de clef étrangère. Il est encore possible de violer une contrainte `NOT NULL` sur une colonne de clef étrangère en effaçant des objets dans le mauvais ordre, par exemple si vous effacez le parent, mais oubliez d'effacer les enfants.

10.9. Réplication d'objets entre deux entrepôts de données

Il est occasionnellement utile de pouvoir prendre un graphe d'instances persistantes et de les rendre persistantes dans un entrepôt différent, sans régénérer les valeurs des identifiants.

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

Le `ReplicationMode` détermine comment `replicate()` traitera les conflits avec des lignes existantes dans la base de données.

- `ReplicationMode.IGNORE` - ignore l'objet s'il y a une ligne existante dans la base de données avec le même identifiant
- `ReplicationMode.OVERWRITE` - écrase n'importe quelle ligne existante dans la base de données avec le même identifiant
- `ReplicationMode.EXCEPTION` - lève une exception s'il y a une ligne dans la base de données avec le même identifiant
- `ReplicationMode.LATEST_VERSION` - écrase la ligne si son numéro de version est plus petit que le numéro de version de l'objet, sinon ignore l'objet

Les cas d'utilisation de cette fonctionnalité incluent la réconciliation de données entrées dans différentes base de données, l'extension des informations de configuration du système durant une mise à jour du produit, retour en arrière sur les changements effectués durant des transactions non-ACID, et plus.

10.10. Flush de la session

De temps en temps la `Session` exécutera les expressions SQL requises pour synchroniser l'état de la connexion JDBC avec l'état des objets retenus en mémoire. Ce processus, *flush*, survient par défaut aux points suivants :

- avant certaines exécutions de requête
- lors d'un appel à `org.hibernate.Transaction.commit()`
- lors d'un appel à `Session.flush()`

Les expressions SQL sont effectuées dans l'ordre suivant :

1. insertion des entités, dans le même ordre que celui des objets correspondants sauvegardés par l'appel à `Session.save()`
2. mise à jour des entités
3. suppression des collections
4. suppression, mise à jour et insertion des éléments des collections
5. insertion des collections
6. suppression des entités, dans le même ordre que celui des objets correspondants qui ont été supprimés par l'appel de `Session.delete()`

Une exception est que des objets utilisant la génération `native` d'identifiants sont insérés lorsqu'ils sont sauvegardés.

Excepté lorsque vous appelez `flush()` explicitement, il n'y a absolument aucune garantie à propos de *quand* la `Session` exécute les appels JDBC, seulement sur l'*ordre* dans lequel ils sont exécutés. Cependant, Hibernate garantit que `Query.list(..)` ne retournera jamais de données périmées, ni des données fausses.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time when the Hibernate `Transaction`

API is used, flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time (see [Section 12.3.2, « Les sessions longues et le versionnage automatique. »](#)).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in [Chapitre 12, Transactions et Accès concurrents](#).

10.11. Persistence transitive

Il est assez pénible de sauvegarder, supprimer, ou rattacher des objets un par un, surtout si vous traitez un graphe d'objets associés. Un cas courant est une relation parent/enfant. Considérez l'exemple suivant :

Si les enfants de la relation parent/enfant étaient des types de valeur (par exemple, une collection d'adresses ou de chaînes de caractères), leur cycle de vie dépendrait du parent et aucune action ne serait requise pour "cascader" facilement les changements d'état. Si le parent est sauvegardé, les objets enfants de type de valeur sont sauvegardés également, si le parent est supprimé, les enfants sont supprimés, etc. Ceci fonctionne même pour des opérations telles que la suppression d'un enfant de la collection ; Hibernate le détectera et étant donné que les objets de type de valeur ne peuvent pas avoir de références partagées, il supprimera l'enfant de la base de données.

Maintenant considérez le même scénario avec un parent dont les objets enfants sont des entités, et non des types de valeur (par exemple, des catégories et des objets, ou un parent et des chatons). Les entités ont leur propre cycle de vie, supportent les références partagées (donc supprimer une entité de la collection ne signifie pas qu'elle peut être supprimée), et il n'y a par défaut pas de cascade d'état d'une entité vers n'importe quelle entité associée. Hibernate n'implémente pas la *persistence par accessibilité* par défaut.

Pour chaque opération basique de la session Hibernate - incluant `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` - il y a un style de cascade correspondant. Respectivement, les styles de cascade s'appellent `persist`, `merge`,

`save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`. Si vous voulez qu'une opération soit cascadée le long d'une association, vous devez l'indiquer dans le document de mappage. Par exemple :

```
<one-to-one name="person" cascade="persist"/>
```

Les styles de cascade peuvent être combinés :

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

Vous pouvez même utiliser `cascade="all"` pour spécifier que *toutes* les opérations devraient être cascadées le long de l'association. La valeur par défaut `cascade="none"` spécifie qu'aucune opération ne sera cascadée.

Un style de cascade spécial, `delete-orphan`, s'applique seulement aux associations un-à-plusieurs, et indique que l'opération `delete()` devrait être appliquée à tout enfant supprimé de l'association.

Recommandations :

- Cela n'a généralement aucun sens d'activer la cascade sur une association `<many-to-one>` ou `<many-to-many>`. Les cascades sont souvent utiles pour des associations `<one-to-one>` et `<one-to-many>`.
- Si la durée de vie de l'objet enfant est liée à la durée de vie de l'objet parent, faites-en un *objet du cycle de vie* en spécifiant `cascade="all,delete-orphan"`.
- Sinon, vous pourriez ne pas avoir besoin de cascade du tout. Mais si vous pensez que vous travaillerez souvent avec le parent et les enfants ensemble dans la même transaction, et que vous voulez vous éviter quelques frappes, considérez l'utilisation de `cascade="persist,merge,save-update"`.

Mapper une association (soit une simple association valuée, soit une collection) avec `cascade="all"` marque l'association comme une relation de style *parent/enfant* où la sauvegarde/mise à jour/suppression du parent entraîne la sauvegarde/mise à jour/suppression de l'enfant ou des enfants.

Par ailleurs, une simple référence à un enfant d'un parent persistant aura pour conséquence la sauvegarde/mise à jour de l'enfant. Cette métaphore est cependant incomplète. Un enfant qui devient non référencé par son parent *n'est pas* automatiquement supprimé, sauf dans le cas d'une association `<one-to-many>` mappée avec `cascade="delete-orphan"`. La sémantique précise des opérations de cascade pour une relation parent/enfant est la suivante :

- Si un parent est passé à `persist()`, tous les enfants sont passés à `persist()`
- Si un parent est passé à `merge()`, tous les enfants sont passés à `merge()`

- Si un parent est passé à `save()`, `update()` ou `saveOrUpdate()`, tous les enfants sont passés à `saveOrUpdate()`
- Si un enfant détaché ou éphémère devient référencé par un parent persistant, il est passé à `saveOrUpdate()`
- Si un parent est supprimé, tous les enfants sont passés à `delete()`
- Si un enfant est déréférencé par un parent persistant, *rien de spécial n'arrive* - l'application devrait explicitement supprimer l'enfant si nécessaire - à moins que `cascade="delete-orphan"` soit paramétré, auquel cas l'enfant "orphelin" est supprimé.

Enfin, la cascade des opérations peut être effectuée sur un graphe donné lors de l'*appel de l'opération* or lors du *flush* suivant. Toutes les opérations, lorsqu'elles sont cascadées, le sont sur toutes les entités associées accessibles lorsque l'opération est exécutée. Cependant `save-update` et `delete-orphan` sont cascadés à toutes les entités associées accessibles lors du flush de la Session.

10.12. Utilisation des méta-données

Hibernate requiert un modèle de méta-niveau très riche de toutes les entités et types valués. De temps en temps, ce modèle est très utile à l'application elle même. Par exemple, l'application pourrait utiliser les méta-données de Hibernate pour implémenter un algorithme de copie en profondeur "intelligent" qui comprendrait quels objets devraient être copiés (par exemple les types de valeur mutables) et lesquels ne devraient pas l'être (par exemple les types de valeurs immutables et, éventuellement, les entités associées).

Hibernate expose les méta-données via les interfaces `ClassMetadata` et `CollectionMetadata` et la hiérarchie `Type`. Les instances des interfaces de méta-données peuvent être obtenues à partir de la `SessionFactory`.

```
Cat fritz = .....;
ClassMetadata catMeta = sessionFactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Read-only entities



Important

Hibernate's treatment of *read-only* entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- Hibernate does not dirty-check the entity's simple properties or single-ended associations;
- Hibernate will not update simple properties or updatable single-ended associations;
- Hibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, Hibernate treats read-only entities the same as entities that are not read-only:

- Hibernate cascades operations to associations as defined in the entity mapping.
- Hibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see [Section 11.2, « Read-only affect on property type »](#).

For details about how to make entities read-only, see [Section 11.1, « Making persistent entities read-only »](#)

Hibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

11.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

Hibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as *immutable*; when an entity of an immutable class is made persistent, Hibernate automatically makes it read-only. see [Section 11.1.1, « Entities of immutable classes »](#) for details
- you can change a default so that entities loaded into the session by Hibernate are automatically made read-only; see [Section 11.1.2, « Loading persistent entities as read-only »](#) for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, scrolls, or iterates, are automatically made read-only; see [Section 11.1.3, « Loading read-only entities from an HQL query/criteria »](#) for details
- you can make a persistent entity that is already in the in the session read-only; see [Section 11.1.4, « Making a persistent entity read-only »](#) for details

11.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, Hibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

Hibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that Hibernate will not allow an entity of an immutable class to be changed so it is not read-only.

11.1.2. Loading persistent entities as read-only



Note

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so Hibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.setDefaultReadOnly( true );
```

To change the default back so entities loaded by Hibernate are not made read-only, call:

```
Session.setDefaultReadOnly( false );
```

You can determine the current setting by calling:

```
Session.isDefaultReadOnly();
```

If `Session.isDefaultReadOnly()` returns `true`, entities loaded by the following are automatically made read-only:

- `Session.load()`
- `Session.get()`
- `Session.merge()`
- executing, scrolling, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see [Section 11.1.3, « Loading read-only entities from an HQL query/criteria »](#)

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via `Session.refresh()`; a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via `Session.persist()`, `Session.save()`, and `Session.update()` `Session.saveOrUpdate()`

11.1.3. Loading read-only entities from an HQL query/criteria



Note

Entities of immutable classes are automatically loaded as read-only.

If `Session.isDefaultReadOnly()` returns `false` (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.setReadOnly( true );
```

`Query.setReadOnly(true)` must be called before `Query.list()`, `Query.uniqueResult()`, `Query.scroll()`, or `Query.iterate()`

For an HQL criteria, call:

```
Criteria.setReadOnly( true );
```

`Criteria.setReadOnly(true)` must be called before `Criteria.list()`, `Criteria.uniqueResult()`, or `Criteria.scroll()`

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if `Session.isDefaultReadOnly()` returns true.

Using `Query.setReadOnly(true)` or `Criteria.setReadOnly(true)` works well when a single HQL query or criteria loads all the entities and initializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
    ( Contract ) session.createQuery(
        "from Contract where customerName = 'Sherman'" )
        .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
session.close();
```

If `Session.isDefaultReadOnly()` returns true, then you can use `Query.setReadOnly(false)` and `Criteria.setReadOnly(false)` to override this session setting and load entities that are not read-only.

11.1.4. Making a persistent entity read-only



Note

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.setReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.setReadOnly(entityOrProxy, false)
```



Important

When a read-only entity or proxy is changed so it is no longer read-only, Hibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
session.refresh( entity );
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.evict( entity );

// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

11.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

Tableau 11.1. Affect of read-only entity on property types

Property/Association Type	Changes flushed to DB?
Simple	no*

Property/Association Type	Changes flushed to DB?
(Section 11.2.1 , « <i>Simple properties</i> »)	
Unidirectional one-to-one	no*
Unidirectional many-to-one	no*
(Section 11.2.2.1 , « <i>Unidirectional one-to-one and many-to-one</i> »)	
Unidirectional one-to-many	yes
Unidirectional many-to-many	yes
(Section 11.2.2.2 , « <i>Unidirectional one-to-many and many-to-many</i> »)	
Bidirectional one-to-one	only if the owning entity is not read-only*
(Section 11.2.3.1 , « <i>Bidirectional one-to-one</i> »)	
Bidirectional one-to-many/many-to-one	only added/removed entities that are not read-only*
inverse collection	yes
non-inverse collection	
(Section 11.2.3.2 , « <i>Bidirectional one-to-many/many-to-one</i> »)	
Bidirectional many-to-many	yes
(Section 11.2.3.3 , « <i>Bidirectional many-to-many</i> »)	

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

11.2.1. Simple properties

When a persistent object is read-only, Hibernate does not dirty-check simple properties.

Hibernate will not synchronize simple property state changes to the database. If you have automatic versioning, Hibernate will not increment the version if any simple properties change.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
```

```

contract.setCustomerName( "Yogi" );
tx.commit();

tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();

```

11.2.2. Unidirectional associations

11.2.2.1. Unidirectional one-to-one and many-to-one

Hibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term *unidirectional single-ended association* when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

Hibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.



Note

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, Hibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```

// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );

```

```
session.setReadOnly( contract, true );
contract.setPlan( null );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

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

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan"
contract.setPlan( newPlan );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
newPlan = ( Plan ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

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

11.2.2.2. Unidirectional one-to-many and many-to-many

Hibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in the collection if they dirty the owning entity.

11.2.3. Bidirectional associations

11.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- Hibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, Hibernate will not increment the version due to local changes to the association.



Note

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, Hibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

11.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

11.2.3.3. Bidirectional many-to-many

Hibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

Transactions et Accès concurrents

L'un des principaux avantages du mécanisme de contrôle des accès concurrents de Hibernate est qu'il est très facile à comprendre. Hibernate utilise directement les connexions JDBC ainsi que les ressources JTA sans y ajouter davantage de mécanisme de blocage. Nous vous recommandons de vous familiariser avec les spécifications JDBC, ANSI et d'isolement de transaction du système de gestion de la base de données que vous utilisez.

Hibernate ne verrouille pas vos objets en mémoire. Votre application peut suivre le comportement défini par le niveau d'isolation de vos transactions de base de données. Notez que grâce à la `Session`, qui est aussi un cache de portée de transaction, Hibernate fournit des lectures répétées pour les recherches par identifiants et les requêtes d'entités (ne rapporte pas les requêtes qui retournent des valeurs scalaires).

En plus du versioning, pour le contrôle automatique optimiste de concurrence, Hibernate fournit également une API (mineure) pour le verrouillage pessimiste des lignes, en générant une syntaxe `SELECT FOR UPDATE`. Le contrôle de concurrence optimiste et cette API seront approfondis ultérieurement dans ce chapitre.

Nous abordons la gestion des accès concurrents en discutant de la granularité des objets `Configuration`, `SessionFactory`, et `Session`, ainsi que des transactions de la base de données et des longues transactions applicatives.

12.1. Portées des sessions et des transactions

Il est important de savoir qu'un objet `SessionFactory` est un objet complexe et optimisé pour fonctionner avec les threads (thread-safe). Il est coûteux à créer et est ainsi prévu pour n'être instancié qu'une seule fois via une instance `Configuration` en général au démarrage de l'application.

Une `Session` n'est pas coûteuse, et c'est un objet non-threadsafe qui ne devrait être utilisé qu'une seule fois pour une requête unique, une conversation, une unité de travail unique et devrait être relâché ensuite. Un objet `Session` ne tentera pas d'obtenir une `ConnectionJDBC` (ou une `Datasource`) si ce n'est pas nécessaire, par conséquent il ne consommera pas de ressource jusqu'à son utilisation.

Afin de compléter ce tableau, vous devez également penser aux transactions de base de données. Une transaction de base de données doit être aussi courte que possible afin de réduire les risques de contention de verrou dans la base de données. De longues transactions à la base de données nuiront à l'extensibilité de vos applications lorsque confrontées à de hauts niveaux de charge. Par conséquent, ce n'est un bon design que de maintenir une transaction ouverte pendant la durée de réflexion de l'utilisateur, jusqu'à ce que l'unité de travail soit achevée.

Quelle est la portée d'une unité de travail? Est-ce qu'une `Session` unique de Hibernate peut avoir une durée de vie dépassant plusieurs transactions à la base de données, ou bien est-ce une

relation un-à-un des portées? Quand faut-il ouvrir et fermer une `Session` et comment définir les démarcations de vos transactions à la base de données ?

12.1.1. Unité de travail

First, let's define a unit of work. A unit of work is a design pattern described by Martin Fowler as « [maintaining] a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. »[PoEAA] In other words, its a series of operations we wish to carry out against the database together. Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions (see [Section 12.1.2, « Longue conversation »](#)). So really we are talking about a more abstract notion of a transaction. The term "business transaction" is also sometimes used in lieu of unit of work.

Tout d'abord n'utilisez pas l'anti-pattern *session-par-operation* et n'ouvrez et ne fermez pas la `Session` à chacun de vos accès simples à la base de données dans un même thread ! Par conséquent, le même raisonnement est applicable à la gestion des transactions dans la base de données. Dans une application, les appels à la base de données doivent être effectués selon une séquence ordonnée et sont regroupés en unités de travail atomiques. (Notez que l'utilisation d'une connexion auto-commit après chaque déclaration SQL est inutile dans une application. Ce mode de fonctionnement existe pour les applications émettant des commandes SQL ad-hoc à partir d'une console. Hibernate désengage le mode auto-commit et s'attend à ce qu'un serveur d'applications le fasse également.) Les transactions avec la base de données ne sont jamais optionnelles. Toute communication avec une base de données doit se dérouler dans une transaction, peu importe si vous lisez ou écrivez des données. Comme déjà mentionné, le comportement auto-commit pour lire les données devrait être évité, puisque plusieurs petites transactions ne seront jamais aussi efficaces qu'une seule plus grosse clairement définie comme unité de travail. Ce dernier choix est de plus beaucoup plus facile à maintenir et plus extensible.

Le modèle d'utilisation le plus fréquemment rencontré dans des applications clients serveur multi-usagers est la *session-par-requête*. Dans ce modèle, la requête d'un client est envoyée au serveur (où la couche de persistance est implémentée via Hibernate), une nouvelle `Session` Hibernate est ouverte et toutes les opérations d'accès à la base de données sont exécutées à l'intérieur de celle-ci. Lorsque le travail est terminé (et que les réponses à envoyer au client ont été préparées), la session est flushée et fermée. Une seule transaction à la base de données peut être utilisée pour répondre à la requête du client. La transaction est démarrée et validée au même moment où la `Session` est ouverte et fermée. La relation entre la `Session` et la `Transaction` est donc un-à-un. Cette relation entre les deux est un-à-un et ce modèle permet de répondre parfaitement aux attentes de la grande majorité des applications.

Le défi réside dans l'implémentation. Hibernate fournit une fonction de gestion intégrée de la "session courante" pour simplifier ce pattern. Il vous suffit de démarrer une transaction lorsqu'une requête est traitée par le serveur, et la terminer avant que la réponse ne soit envoyée au client. Vous pouvez choisir la manière de l'effectuer, les solutions communes sont un `ServletFilter`, l'interception via AOP avec une coupe transverse (pointcut) sur les méthodes de type "service", ou un conteneur avec interception/proxy. Un conteneur EJB est un moyen standard d'implémenter ce genre d'aspect transverse comme la démarcation des transactions sur les EJB session, de

manière déclarative avec CMT. Si vous décidez d'utiliser la démarcation programmatique des transactions, préférez l'API Hibernate `Transaction` traitée plus tard dans ce chapitre, afin de faciliter l'utilisation et la portabilité du code.

Your application code can access a "current session" to process the request by calling `sessionFactory.getCurrentSession()`. You will always get a `Session` scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see [Section 2.5, « Sessions contextuelles »](#).

Il est parfois utile d'étendre la portée d'une `Session` et d'une transaction à la base de données jusqu'à ce que "la vue soit rendue". Ceci est particulièrement utile dans des applications à base de servlet qui utilisent une phase de rendu séparée une fois que la réponse a été préparée. Étendre la transaction avec la base de données jusqu'à la fin du rendering de la vue est aisé si vous implémentez votre propre intercepteur. Cependant, ce n'est pas facile si vous vous appuyez sur les EJB avec CMT, puisqu'une transaction sera achevée au retour de la méthode EJB, avant le rendu de la vue. Rendez vous sur le site Hibernate et sur le forum pour des astuces et des exemples sur le pattern *Open Session in View*.

12.1.2. Longue conversation

Le paradigme "session-per-request" n'est pas le seul élément à utiliser dans le design de vos unités de travail. Plusieurs processus d'affaire requièrent toute une série d'interactions avec l'utilisateur, entrelacées d'accès à la base de donnée. Dans une application Web ou une application d'entreprise, il serait inacceptable que la durée de vie d'une transaction s'étale sur plusieurs interactions avec l'usager. Considérez l'exemple suivant :

- Un écran s'affiche. Les données vues par l'usager ont été chargées dans l'instance d'un objet `Session`, dans le cadre d'une transaction de base de données. L'usager est libre de modifier ces objets.
- L'usager clique "Sauvegarder" après 5 minutes et souhaite persister les modifications qu'il a apportées. Il s'attend à être la seule personne à avoir modifié ces données et qu'aucune modification conflictuelle ne se soit produite durant ce laps de temps.

Ceci s'appelle une unité de travail. Du point de vue de l'utilisateur: une *conversation* (ou *transaction d'application*). Il y a plusieurs façon de mettre ceci en place dans votre application.

Une première implémentation naïve pourrait consister à garder la `Session` et la transaction à la base de données ouvertes durant le temps de travail de l'usager, à maintenir les enregistrements verrouillés dans la base de données afin d'éviter des modifications concurrentes et de maintenir l'isolation et l'atomicité de la transaction de l'usager. Ceci est un anti-pattern à éviter, puisque le verrouillage des enregistrements dans la base de données ne permettrait pas à l'application de gérer un grand nombre d'usagers concurrents.

Il apparaît donc évident qu'il faille utiliser plusieurs transactions BDD afin d'implémenter la conversation. Dans ce cas, maintenir l'isolation des processus d'affaire devient partiellement la

responsabilité de la couche applicative. Ainsi, la durée de vie d'une conversation devrait englober celle d'une ou de plusieurs transactions de base de données. Celle-ci sera atomique seulement si l'écriture des données mises à jour est faite exclusivement par la dernière transaction BDD la composant. Toutes les autres sous transactions BD ne doivent faire que la lecture de données. Ceci est relativement facile à mettre en place, surtout avec l'utilisation de certaines fonctionnalités d'Hibernate :

- *Versionnage Automatique* - Hibernate peut gérer automatiquement les accès concurrents de manière optimiste et détecter si une modification concurrente s'est produite durant le temps de réflexion d'un usager. A vérifier en fin de conversation.
- *Objets Détachés* - Si vous décidez d'utiliser le paradigme *session-par-requête* discuté plus haut, toutes les entités chargées en mémoire deviendront des objets détachés durant le temps de réflexion de l'usager. Hibernate vous permet de rattacher ces objets et de persister les modifications y ayant été apportées. Ce pattern est appelé: *session-per-request-with-detached-objects* (littéralement: session- par-requête-avec-objets-détachés). Le versionnage automatique est utilisé afin d'isoler les modifications concurrentes.
- *Session Longues (conversation)* - Une *Session* Hibernate peut être déconnectée de la couche JDBC sous-jacente après que `commit()` ait été appelé sur une transaction à la base de données et reconnectée lors d'une nouvelle requête-client. Ce pattern s'appelle: *session-per-conversation* (Littéralement: session-par- conversation) et rend superflu le rattachement des objets. Le versionnage automatique est utilisé afin d'isoler les modifications concurrentes.

Les deux patterns *session-per-request-with-detached-objects* (session-par-requête-avec-objets-détachés) et *session-per-conversation* (session-par-conversation) ont chacun leurs avantages et désavantages qui seront exposés dans ce même chapitre, dans la section au sujet du contrôle optimiste de concurrence.

12.1.3. L'identité des objets

Une application peut accéder à la même entité persistante de manière concurrente dans deux *Sessions* différentes. Toutefois, une instance d'une classe persistante n'est jamais partagée par deux instances distinctes de la classe *Session*. Il existe donc deux notions de l'identité d'un objet :

Identité de database

```
foo.getId().equals( bar.getId() )
```

Identité JVM

```
foo==bar
```

Ainsi, pour des objets attachés à une *Session particulière* (c'est-à-dire dans la portée d'une instance de *Session*), ces deux notions d'identité sont équivalentes et l'identité JVM pour l'identité de la base de données sont garanties par Hibernate. Cependant, alors qu'une application peut

accéder de manière concurrente au "même" objet métier (identité persistante) dans deux sessions différentes, les deux instances seront en fait "différentes" (en ce qui a trait à l'identité JVM). Les conflits sont résolus automatiquement par approche optimiste grâce au système de versionnage automatique au moment du flush/sauvegarde.

Cette approche permet de reléguer à Hibernate et à la base de données sous-jacente le soin de gérer les problèmes d'accès concurrents. Cette manière de faire assure également une meilleure extensibilité de l'application puisque assurer l'identité JVM dans un thread ne nécessite pas de mécanismes de verrouillage coûteux ou d'autres dispositifs de synchronisation. Une application n'aura jamais besoin de synchroniser des objets d'affaire tant qu'elle peut garantir qu'un seul thread aura accès à une instance de `Session`. Dans le cadre d'exécution d'un objet `Session`, l'application peut utiliser en toute sécurité `==` pour comparer des objets.

Une application qui utiliserait `==` à l'extérieur du cadre d'exécution d'une `Session` pourrait obtenir des résultats inattendus. Par exemple, si vous mettez deux objets dans le même `Set`, ceux-ci pourraient avoir la même identité de base de données (c'est-à-dire ils représentent le même enregistrement), mais leur identité JVM pourrait être différente (elle ne peut, par définition, pas être garantie sur deux objets détachés). Le développeur doit donc redéfinir l'implémentation des méthodes `equals()` et `hashCode()` dans les classes persistantes et y adjoindre sa propre notion d'identité. Il existe toutefois une restriction : il ne faut jamais utiliser uniquement l'identifiant de la base de données dans l'implémentation de l'égalité ; il faut utiliser une clé d'affaire, généralement une combinaison de plusieurs attributs uniques, si possible immuables. Les identifiants de base de données vont changer si un objet transitoire (transient) devient persistant. Si une instance transitoire (en général avec des instances dégachées) est contenue dans un `Set`, changer le `hashCode` brisera le contrat du `Set`. Les attributs pour les clés d'affaire n'ont pas à être aussi stables que des clés primaires de bases de données. Il suffit simplement qu'elles soient stables tant et aussi longtemps que les objets sont dans le même `Set`. Veuillez consulter le site web Hibernate pour des discussions plus pointues à ce sujet. Notez que ce concept n'est pas propre à Hibernate mais bien général à l'implémentation de l'identité et de l'égalité en Java.

12.1.4. Problèmes communs

Bien qu'il puisse y avoir quelques rares exceptions à cette règle, il est recommandé de ne jamais utiliser les anti-modèles *session-par-utilisateur-session* ou *session-par-application*. Notez que certains des problèmes suivants pourraient néanmoins survenir avec des modèles recommandés, assurez-vous de bien comprendre les implications de chacun des modèles avant de prendre une décision concernant votre design :

- L'objet `Session` n'est pas conçu pour être utilisé par de multiples threads. En conséquence, les objets potentiellement multi-thread comme les requêtes HTTP, les EJB `Session` et `Swing Worker`, risquent de provoquer des conditions de course dans la `Session` si celle-ci est partagée. Si vous gardez votre `Session` Hibernate dans la `HttpSession` (le sujet sera traité ultérieurement), il serait préférable de synchroniser les accès à la session `Http` afin d'éviter qu'un usager ne recharge une page assez rapidement pour que deux requêtes exécutant dans des threads concurrents n'utilisent la même `Session`.

- Lorsque Hibernate lance une exception, le rollback de la transaction en cours dans la base de données, doit être effectué et la `Session` immédiatement fermée. (Nous approfondirons le sujet plus loin) Si votre `Session` est liée à l'application, il faut arrêter l'application. Le rollback de la transaction de base de données ne remettra pas les objets dans leur état du début de la transaction. Ainsi, cela signifie que l'état de la base de données et les objets d'affaires pourraient être désynchronisés d'avec les enregistrements. Généralement, cela ne cause pas de réel problème puisque la plupart des exceptions sont non traitables et de toutes façons, vous devez recommencer le processus après le rollback).
- The `Session` caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an `OutOfMemoryException`. One solution is to call `clear()` and `evict()` to manage the `Session` cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in [Chapitre 14, Traitement par lot](#). Keeping a `Session` open for the duration of a user session also means a higher probability of stale data.

12.2. Démarcation des transactions de base de données

La démarcation des transactions de base de données (ou système) est toujours nécessaire. Aucune communication avec la base de données ne peut être effectuée à l'extérieur du cadre d'une transaction. (Il semble que ce concept soit mal compris par plusieurs développeurs trop habitués à utiliser le mode auto-commit.) Utilisez toujours la démarcation des des transactions, même pour des opérations en lecture seule. Certains niveaux d'isolation et certaines possibilités offertes par les bases de données permettent de l'éviter, il n'est jamais désavantageux de toujours explicitement indiquer les bornes de transaction. Il est certain qu'une transaction unique de base de données sera plus performante que de nombreuses petites transactions, même pour les opérations simples de lecture.

Une application utilisant Hibernate peut s'exécuter dans un environnement léger n'offrant pas la gestion automatique des transactions (application autonome, application web simple ou applications Swing) ou dans un environnement J2EE offrant des services de gestion automatiques des transactions JTA. Dans un environnement simple, Hibernate a généralement la responsabilité de la gestion de son propre pool de connexions à la base de données. Le développeur de l'application doit manuellement délimiter les transactions. En d'autres mots, il appartient au développeur de gérer les appels à `Transaction.begin()`, `Transaction.commit()` et `Transaction.rollback()`. Un environnement transactionnel J2EE (serveur d'application J2EE) doit offrir la gestion des transactions au niveau du conteneur J2EE. Les bornes de transaction peuvent normalement être définies de manière déclarative dans les descripteurs de déploiement d'EJB Session, par exemple. La gestion programmatique des transactions n'y est donc plus nécessaire.

Cependant, il est souvent préférable d'avoir une couche de persistance portable entre les environnements non gérés de ressources locales et les systèmes qui s'appuient sur JTA mais utilisent BMT à la place de CMT. Dans les deux cas, vous utiliserez la démarcation de transaction

programmatique. Hibernate offre donc une API appelée `Transaction` qui sert d'enveloppe pour le système de transaction natif de l'environnement de déploiement. Il n'est pas obligatoire d'utiliser cette API, mais il est fortement conseillé de le faire, sauf lors de l'utilisation de CMT Session Bean.

Il existe quatre étapes distinctes lors de la fermeture d'une `Session` :

- flush de la session
- commit de la transaction
- fermeture de la session
- gestion des exceptions

La notion de "Flushing" a déjà été expliquée, nous abordons maintenant la démarcation des transactions et la gestion des exceptions dans les environnements gérés et non-gérés.

12.2.1. Environnement non gérés

Si la couche de persistance Hibernate s'exécute dans un environnement non géré, les connexions à la base de données seront généralement prises en charge par le mécanisme de pool d'Hibernate qui obtient les connexions. La gestion de la session et de la transaction se fera donc de la manière suivante :

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

You do not have to `flush()` the `Session` explicitly: the call to `commit()` automatically triggers the synchronization depending on the [FlushMode](#) for the session. A call to `close()` marks the end of a session. The main implication of `close()` is that the JDBC connection will be relinquished by the session. This Java code is portable and runs in both non-managed and JTA environments.

Une solution plus flexible est la gestion par contexte de la session courante intégrée, fournie par Hibernate que nous avons déjà rencontrée :

```
// Non-managed environment idiom with getCurrentSession()
try {
```

```
factory.getCurrentSession().beginTransaction();

// do some work
...

factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

Vous ne verrez probablement jamais ces exemples de code dans les applications ; les exceptions fatales (exceptions du système) ne devraient être traitées que dans la couche la plus "haute". En d'autres termes, le code qui exécute les appels à Hibernate (à la couche de persistance) et le code qui gère les `RuntimeException` (qui ne peut généralement effectuer qu'un nettoyage et une sortie) sont dans des couches différentes. La gestion du contexte courant par Hibernate peut simplifier notablement ce design, puisqu'il vous suffit d'accéder à la `SessionFactory`. La gestion des exceptions est traitée plus loin dans ce chapitre.

Notez que vous devriez sélectionner `org.hibernate.transaction.JDBCTransactionFactory` (le défaut), pour le second exemple "thread" comme votre `hibernate.current_session_context_class`.

12.2.2. Utilisation de JTA

Si votre couche de persistance s'exécute dans un serveur d'applications (par exemple, derrière un EJB Session Bean), toutes les datasources utilisées par Hibernate feront automatiquement partie de transactions JTA globales. Vous pouvez également installer une implémentation autonome JTA et l'utiliser sans l'EJB. Hibernate propose deux stratégies pour réussir l'intégration JTA.

Si vous utilisez des transactions gérées par un EJB (bean managed transactions - BMT), Hibernate informera le serveur d'applications du début et de la fin des transactions si vous utilisez l'API `Transaction`. Ainsi, le code de gestion des transactions sera identique dans les environnements non gérés.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
```

```
finally {
    sess.close();
}
```

Si vous souhaitez utiliser une `Session` couplée à la transaction, c'est à dire, utiliser la fonctionnalité `getCurrentSession()` pour la propagation facile du contexte, vous devez utiliser l'API JTA `UserTransaction` directement :

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
        .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

Avec CMT, la démarcation des transactions est faite dans les descripteurs de déploiement des Beans Sessions et non de manière programmatique, par conséquent le code est réduit à :

```
// CMT idiom
Session sess = factory.getCurrentSession();

// do some work
...
```

Dans un EJB CMT, le rollback aussi intervient automatiquement, puisqu'une `RuntimeException` non traitée et soulevée par une méthode d'un bean session indique au conteneur d'annuler la transaction globale. *Ceci veut donc dire que vous n'avez pas à utiliser l'API `Transaction` de Hibernate dans CMT ou BMT et vous obtenez la propagation automatique de la session courante liée à la transaction.*

Notez que le fichier de configuration Hibernate devrait contenir les valeurs `org.hibernate.transaction.JTATransactionFactory` dans un environnement BMT ou `org.hibernate.transaction.CMTTransactionFactory` dans un environnement CMT là où vous configurez votre fabrique de transaction Hibernate. N'oubliez pas non plus de spécifier le paramètre `org.hibernate.transaction.manager_lookup_class`. De plus, assurez vous de fixer votre `hibernate.current_session_context_class` soit à `"jta"` ou de ne pas le configurer (compatibilité avec les versions précédentes).

La méthode `getCurrentSession()` a un inconvénient dans les environnements JTA. Il y a une astuce qui est d'utiliser un mode de libération de connexion `after_statement`, qui est alors utilisé par défaut. Du à une étrange limitation de la spec JTA, il n'est pas possible à Hibernate de nettoyer automatiquement un `ScrollableResults` ouvert ou une instance d'`Iterator` retournés `scroll()` ou `iterate()`. Vous devez libérer le curseur base de données sous jacent ou invoquer `Hibernate.close(Iterator)` explicitement depuis un bloc `finally`. (Bien sur, la plupart des applications peuvent éviter d'utiliser `scroll()` ou `iterate()` dans un code JTA ou CMT.)

12.2.3. Gestion des exceptions

Si une `Session` lance une exception (incluant les exceptions du type `SQLException` ou d'un sous-type), vous devez immédiatement effectuer le rollback de la transaction, appeler `Session.close()` et relâcher les références sur l'objet `Session`. La `Session` contient des méthodes pouvant la mettre dans un état inutilisable. Vous devez considérer qu'*aucune* exception lancée par Hibernate n'est traitable comme recouvrable. Assurez-vous de fermer la session en appelant `close()` dans un bloc `finally`.

L'exception `HibernateException`, qui englobe la plupart des exceptions pouvant survenir dans la couche de persistance Hibernate, est une exception non vérifiée (Ceci n'était pas le cas dans des versions antérieures de Hibernate.) Nous pensons que nous ne devrions pas forcer un développeur à gérer une exception qu'il ne peut de toute façon pas traiter dans une couche technique. Dans la plupart des systèmes, les exceptions non vérifiées et les exceptions fatales sont gérées en amont du processus (dans les couches hautes) et un message d'erreur est alors affiché à l'utilisateur (ou un traitement alternatif est invoqué.) Veuillez noter que Hibernate peut également lancer des exceptions non vérifiées d'un autre type que `HibernateException`. Celles-ci sont également non traitables et vous devez les traiter comme telles.

Hibernate englobe les `SQLException`s lancées lors des interactions directes avec la base de données dans des exceptions de type: `JDBCException`. En fait, Hibernate essaiera de convertir l'exception dans un sous-type plus significatif de `JDBCException`. L'exception `SQLException` sous-jacente est toujours disponible via la méthode `JDBCException.getCause()`. Hibernate convertit le `SQLExceptionConverter` en une sous-classe `JDBCException`, en utilisant le `SQLExceptionConverter` qui est rattaché à l'objet `SessionFactory`. Par défaut, le `SQLExceptionConverter` est défini par le dialecte configuré dans Hibernate. Toutefois, il est possible de fournir sa propre implémentation de l'interface. (Veuillez vous référer à la javadoc sur la classe `SQLExceptionConverterFactory` pour plus de détails. Les sous-types standard de `JDBCException` sont :

- `JDBCConnectionException` - indique une erreur de communication avec la couche JDBC sous-jacente.
- `SQLGrammarException` - indique un problème de grammaire ou de syntaxe avec la requête SQL envoyée.
- `ConstraintViolationException` - indique une violation de contrainte d'intégrité.
- `LockAcquisitionException` - indique une erreur de verrouillage lors de l'exécution de la requête.

- `GenericJDBCException` - indique une erreur générique ne correspondant à aucune autre catégorie.

12.2.4. Timeout de transaction

Une des caractéristiques extrêmement importante fournie dans les environnements gérés tels les EJB, est la gestion du timeout de transaction qui n'est jamais fournie pour le code non géré. La gestion des dépassements de temps de transaction vise à s'assurer qu'une transaction agissant incorrectement ne viendra pas bloquer indéfiniment les ressources de l'application et ne retourner aucune réponse à l'utilisateur. Hibernate ne peut fournir cette fonctionnalité dans un environnement transactionnel non-JTA. Par contre, Hibernate gère les opérations d'accès aux données en allouant un temps maximal aux requêtes pour s'exécuter. Ainsi, une requête créant de l'inter blocage ou retournant de très grandes quantités d'informations pourrait être interrompue. Dans un environnement géré, Hibernate peut déléguer au gestionnaire de transaction JTA, le soin de gérer les dépassements de temps. Cette fonctionnalité est abstraite par l'objet `Transaction`.

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Notez que `setTimeout()` ne peut pas être appelé d'un EJB CMT, puisque le timeout des transaction doit être spécifié de manière déclarative.

12.3. Contrôle de concurrence optimiste

La gestion optimiste des accès concurrents avec versionnage est la seule approche pouvant garantir l'extensibilité des applications à haut niveau de charge. Le système de versionnage utilise des numéros de version ou l'horodatage pour détecter les mise à jour causant des conflits avec d'autres actualisations antérieures (et pour éviter la perte de mise à jour). Hibernate propose trois approches possibles pour l'écriture de code applicatif utilisant la gestion optimiste d'accès concurrents. Le cas d'utilisation décrit plus bas fait mention de longues conversations, mais le versionnage peut également améliorer la qualité d'une application en prévenant la perte de mise à jour dans les transactions uniques de base de données.

12.3.1. Vérification du versionnage au niveau applicatif

Dans cet exemple d'implémentation utilisant peu les fonctionnalités de Hibernate, chaque interaction avec la base de données se fait en utilisant une nouvelle `Session` et le développeur doit recharger les données persistantes à partir de la base de données avant de les manipuler. Cette implémentation force l'application à vérifier la version des objets afin de maintenir l'isolation transactionnelle. Cette approche, semblable à celle retrouvée pour les EJB, est la moins efficace parmi celles qui sont présentées dans ce chapitre.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty( "bar" );

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

Le mappage de la propriété `version` est fait via `<version>` et Hibernate l'incrémentera automatiquement à chaque `flush()` si l'entité doit être mise à jour.

Bien sûr, si votre application ne fait pas face à beaucoup d'accès concurrents et ne nécessite pas l'utilisation du versionnage, cette approche peut également être utilisée, il n'y a qu'à ignorer le code relié au versionnage. Dans ce cas, la stratégie du *last commit wins* (littéralement: le dernier commit l'emporte) sera utilisée pour les conversations (longues transactions applicatives). Gardez à l'esprit que cette approche pourrait rendre perplexe les utilisateurs de l'application car ils pourraient perdre des données mises à jour sans qu'aucun message d'erreur ne leur soit présenté et sans avoir la possibilité de fusionner les données.

Il est clair que la gestion manuelle de la vérification du versionnage des objets ne peut être effectuée que dans certains cas triviaux et que cette approche n'est pas valable pour la plupart des applications. De manière générale, les applications ne cherchent pas à actualiser de simples objets sans relations, elles le font généralement pour de larges graphes d'objets. Hibernate peut gérer automatiquement la vérification des versions d'objets en utilisant soit une `Session` longue, soit des instances détachées comme paradigme des conversations.

12.3.2. Les sessions longues et le versionnage automatique.

Dans ce scénario, une seule instance de `Session` et des objets persistants est utilisée pour toute la conversation, connue sous *session-par-conversation*. Hibernate vérifie la version des objets persistants avant d'effectuer le `flush()` et lance une exception si une modification concurrente est détectée. Il appartient alors au développeur de gérer l'exception. Les traitements alternatifs généralement proposés sont alors de permettre à l'utilisateur de faire la fusion des données ou de lui offrir de recommencer son travail à partir des données les plus récentes dans la base de données.

Notez que lorsqu'une application est en attente d'une action de la part de l'utilisateur, la `Session` n'est pas connectée à la couche JDBC sous-jacente. C'est la manière la plus efficace de gérer les accès à la base de données. L'application ne devrait pas se préoccuper du versionnage des objets, ou du rattachement des objets détachés, ni du rechargement de tous les objets à chaque transaction.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush(); // Only for last transaction in conversation
t.commit();      // Also return JDBC connection
session.close(); // Only for last transaction in conversation
```

L'objet `foo` sait quel objet `Session` l'a chargé. `Session.reconnect()` obtient une nouvelle connexion (celle-ci peut être également fournie) et permet à la session de continuer son travail. La méthode `Session.disconnect()` déconnecte la session de la connexion JDBC et retourne celle-ci au pool de connexion (à moins que vous ne lui ayez fourni vous même la connexion.) Après la reconnexion, afin de forcer la vérification du versionnage de certaines entités que vous ne cherchez pas à actualiser, vous pouvez faire un appel à `Session.lock()` en mode `LockMode.READ` pour tout objet ayant pu être modifié par une autre transaction. Il n'est pas nécessaire de verrouiller les données que vous désirez mettre à jour. En général, vous configurerez `FlushMode.NEVER` sur une `Session` étendue, de façon que seul le dernier cycle de transaction de la base de données puissent persister toutes les modifications effectuées dans cette conversation. Par conséquent, cette dernière transaction inclura l'opération `flush()`, de même que `close()` la session pour finir la conversation.

Ce modèle peut présenter des problèmes si la `Session` est trop volumineuse pour être stockée entre les actions de l'utilisateur. Plus spécifiquement, une session `HttpSession` se doit d'être la plus petite possible. Puisque la `Session` joue obligatoirement le rôle de mémoire cache de premier niveau et contient à ce titre tous les objets chargés, il est préférable de n'utiliser une `Session` que pour une seule conversation, car les objets risquent d'y être rapidement périmés.



Remarque

Notez que des versions précédentes de Hibernate exigeaient une déconnexion explicite et une reconnexion d'une `Session`. Ces méthodes sont périmées, puisque commencer et terminer une transaction a le même effet.

Notez que la `Session` déconnectée devrait être conservée près de la couche de persistance. Autrement dit, utilisez un EJB stateful pour conserver la `Session` dans un environnement 3 niveaux et évitez de la sérialiser et de la transférer à la couche de présentation (c'est-à-dire qu'il est préférable de ne pas la conserver dans la session `HttpSession`.)

Le modèle de session étendue, ou *session-par-conversation*, est plus difficile à implémenter avec la gestion automatique de contexte de session courante. À cet effet, vous devez fournir votre propre implémentation de `CurrentSessionContext`, pour des exemples consultez [Hibernate Wiki](#).

12.3.3. Les objets détachés et le versionnage automatique

Chaque interaction avec le système de persistance se fait via une nouvelle `Session`. Toutefois, les mêmes instances d'objets persistants sont réutilisées pour chacune de ces interactions. L'application doit pouvoir manipuler l'état des instances détachées ayant été chargées antérieurement via une autre session. Pour ce faire, ces objets persistants doivent être rattachés à la `Session` courante en utilisant `Session.update()`, `Session.saveOrUpdate()`, ou `Session.merge()`.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

Encore une fois, Hibernate vérifiera la version des instances devant être actualisées durant le `flush()`. Une exception sera lancée si des conflits sont détectés.

Vous pouvez également utiliser `lock()` au lieu de `update()` et utiliser le mode `LockMode.READ` (qui lancera une vérification de version, en ignorant tous les niveaux de mémoire cache) si vous êtes certain que l'objet n'a pas été modifié.

12.3.4. Personnaliser le versionnage automatique

Vous pouvez désactiver l'incrémentation automatique du numéro de version de certains attributs et collections en mettant la valeur du paramètre de mapping `optimistic-lock` à `false`. Hibernate cessera ainsi d'incrémenter leur numéro de version si la propriété est dirty.

Certaines entreprises possèdent de vieux systèmes dont les schémas de bases de données sont statiques et ne peuvent être modifiés. Il existe aussi des cas où plusieurs applications doivent accéder à la même base de données, mais certaines d'entre elles ne peuvent gérer les numéros de version ou les champs horodatés. Dans les deux cas, le versionnage ne peut se fier à une colonne particulière dans une table. Afin de forcer la vérification de version dans un système sans en faire le mappage, mais en forçant une comparaison des états de tous les attributs d'une entité, vous pouvez utiliser l'attribut `optimistic-lock="all"` dans le mappage `<class>`. Veuillez noter que cette manière de gérer le versionnage ne peut être utilisée que si l'application utilise de longues sessions, lui permettant de comparer l'ancien état et le nouvel état d'une entité. L'utilisation d'un modèle `session-per-request-with-detached-objects` devient alors impossible.

Il peut être souhaitable de permettre les modifications concurrentes du moment que les modifications ne se chevauchent pas. En configurant la propriété à `optimistic-lock="dirty"` quand vous mappez le `<class>`, Hibernate ne fera la comparaison que des champs devant être actualisés lors du flush.

Dans les deux cas: en utilisant une colonne de version/horodatée ou via la comparaison de l'état complet de l'objet ou de ses champs modifiés, Hibernate ne créera qu'une seule commande `UPDATE` par entité avec la clause `WHERE` appropriée pour vérifier la version et mettre à jour les informations. Si vous utilisez la persistance transitive pour propager l'évènement de rattachement à des entités associées, il est possible que Hibernate génère des commandes de mise à jour inutiles. Ceci n'est généralement pas un problème, mais certains déclencheurs *on update* dans la base de données pourraient être activés même si aucun changement n'était réellement persisté sur des objets détachés. Vous pouvez personnaliser ce comportement en indiquant `select-before-update="true"` dans l'élément de mappage `<class>`. Ceci forcera Hibernate à faire le `SELECT` de l'instance afin de s'assurer que l'entité doit réellement être actualisée avant de lancer la commande de mise à jour de l'enregistrement.

12.4. Verrouillage pessimiste

Il n'est nécessaire de s'attarder à la stratégie de verrouillage des entités dans une application utilisant Hibernate. Il est généralement suffisant de définir le niveau d'isolation pour les connexions JDBC et de laisser ensuite la base de donnée effectuer son travail. Toutefois, certains utilisateurs avancés peuvent vouloir obtenir un verrouillage pessimiste exclusif sur un enregistrement, ou le ré-obtenir au lancement d'une nouvelle transaction.

Hibernate utilisera toujours le mécanisme de verrouillage de la base de données et ne verrouillera jamais les objets en mémoire.

La classe `LockMode` définit les différents niveaux de verrouillage pouvant être obtenus par Hibernate. Le verrouillage est obtenu par les mécanismes suivants :

- `LockMode.WRITE` est obtenu automatiquement quand Hibernate actualise ou insère un enregistrement.
- `LockMode.UPGRADE` peut être obtenu de manière explicite via la requête en utilisant `SELECT ... FOR UPDATE` sur une base de données supportant cette syntaxe.
- `LockMode.UPGRADE_NOWAIT` peut être obtenu de manière explicite en utilisant `SELECT ... FOR UPDATE NOWAIT` sur Oracle.
- `LockMode.READ` est obtenu automatiquement quand Hibernate lit des données dans un contexte d'isolation `Repeatable Read` ou `Serializable`. Peut être ré-obtenu explicitement via une requête d'utilisateur.
- `LockMode.NONE` représente l'absence de verrouillage. Tous les objets migrent vers ce mode à la fin d'une `Transaction`. Les objets associés à une session via un appel à `saveOrUpdate()` commencent également leur cycle de vie dans ce mode verrouillé.

Les requêtes explicites d'utilisateur sont exprimées d'une des manières suivantes :

- Un appel à `Session.load()`, en spécifiant un niveau verrouillage `LockMode`.
- Un appel à `Session.lock()`.
- Une appel à `Query.setLockMode()`.

Si `Session.load()` est appelé avec le paramètre de niveau de verrouillage `UPGRADE` ou `UPGRADE_NOWAIT` et que l'objet demandé n'est pas présent dans la session, celui-ci sera chargé à l'aide d'une requête `SELECT ... FOR UPDATE`. Si la méthode `load()` est appelée pour un objet déjà en session avec un verrouillage moindre que celui demandé, Hibernate appellera la méthode `lock()` pour cet objet.

`Session.lock()` effectue une vérification de version si le niveau de verrouillage est `READ`, `UPGRADE` ou `UPGRADE_NOWAIT`. Dans le cas des niveaux `UPGRADE` ou `UPGRADE_NOWAIT`, une requête `SELECT ... FOR UPDATE` sera utilisée.

Si une base de données ne supporte pas le niveau de verrouillage demandé, Hibernate utilisera un niveau alternatif convenable au lieu de lancer une exception. Ceci assurera la portabilité de vos applications.

12.5. Modes de libération de connexion

Le comportement original (2.x) de Hibernate pour la gestion des connexions JDBC était que la `Session` obtenait une connexion dès qu'elle en avait besoin et la libérait une fois la session fermée. Hibernate 3.x a introduit les modes de libération de connexion pour indiquer à la session comment gérer les transactions JDBC. Notez que la discussion suivante n'est pertinente que pour des connexions fournies par un `ConnectionProvider`, celles gérées par l'utilisateur dépassent l'objectif de cette discussion. Les différents modes de libération sont identifiés par les valeurs énumérées de `org.hibernate.ConnectionReleaseMode`:

- `ON_CLOSE` - est essentiellement le comportement passé décrit ci-dessus. La session Hibernate obtient une connexion lorsqu'elle en a besoin et la garde jusqu'à ce que la session se ferme.
- `AFTER_TRANSACTION` - indique de relâcher la connexion après qu'une `org.hibernate.Transaction` soit achevée.
- `AFTER_STATEMENT` (aussi appelé libération brutale) - indique de relâcher les connexions après chaque exécution d'un statement. Ce relâchement agressif est annulé si ce statement laisse des ressources associées à une session donnée ouvertes, actuellement ceci n'arrive que lors de l'utilisation de `org.hibernate.ScrollableResults`.

Le paramètre de configuration `hibernate.connection.release_mode` est utilisé pour spécifier quel mode de libération doit être utilisé. Les valeurs possibles sont :

- `auto` (valeur par défaut) - ce choix délègue le choix de libération à la méthode `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()`. Pour la `JTATransactionFactory`, elle retourne `ConnectionReleaseMode.AFTER_STATEMENT`; pour `JDBCTransactionFactory`, elle retourne `ConnectionReleaseMode.AFTER_TRANSACTION`. C'est rarement une bonne idée de changer ce comportement par défaut puisque les erreurs

soulevées par ce paramétrage tend à indiquer la présence de bogues et/ou d'erreurs dans le code de l'utilisateur.

- `on_close` - indique d'utiliser `ConnectionReleaseMode.ON_CLOSE`. Ce paramétrage existe pour garantir la compatibilité avec les versions précédentes, mais ne devrait plus être utilisé.
- `after_transaction` - indique d'utiliser `ConnectionReleaseMode.AFTER_TRANSACTION`. Ne devrait pas être utilisé dans les environnements JTA. Notez aussi qu'avec `ConnectionReleaseMode.AFTER_TRANSACTION`, si une session est considérée comme étant en mode auto-commit les connexions seront relâchées comme si le mode était `AFTER_STATEMENT`.
- `after_statement` - indique d'utiliser `ConnectionReleaseMode.AFTER_STATEMENT`. De plus, le `ConnectionProvider` utilisé est consulté pour savoir s'il supporte ce paramétrage (`supportsAggressiveRelease()`). Si ce n'est pas le cas, le mode de libération est ré-initialisé à `ConnectionReleaseMode.AFTER_TRANSACTION`. Ce paramétrage n'est sûr que dans les environnements où il est possible d'obtenir à nouveau la même connexion JDBC à chaque fois que l'on fait un appel de `ConnectionProvider.getConnection()` ou dans les environnements auto-commit où il n'est pas important d'obtenir plusieurs fois la même connexion.

Intercepteurs et événements

Il est souvent utile pour l'application de réagir à certains événements qui surviennent dans Hibernate. Cela autorise l'implémentation de certaines fonctionnalités génériques, et l'extension de fonctionnalités d'Hibernate.

13.1. Intercepteurs

L'interface `Interceptor` fournit des "callbacks" de la session vers l'application permettant à l'application de consulter et/ou de manipuler des propriétés d'un objet persistant avant qu'il soit sauvegardé, mis à jour, supprimé ou chargé. Une utilisation possible de cette fonctionnalité est de tracer l'accès à l'information. Par exemple, l'Interceptor suivant positionne `createTimestamp` quand un `Auditable` est créé et met à jour la propriété `lastUpdateTimestamp` quand un `Auditable` est mis à jour.

Vous pouvez soit implémenter `Interceptor` directement ou (mieux) étendre `EmptyInterceptor`.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                              Serializable id,
                              Object[] currentState,
                              Object[] previousState,
                              String[] propertyNames,
                              Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
```

```
        currentState[i] = new Date();
        return true;
    }
}
}
return false;
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {
    if ( entity instanceof Auditable ) {
        loads++;
    }
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if ( entity instanceof Auditable ) {
        creates++;
        for ( int i=0; i<propertyNames.length; i++ ) {
            if ( "createTimestamp".equals( propertyNames[i] ) ) {
                state[i] = new Date();
                return true;
            }
        }
    }
    return false;
}

public void afterTransactionCompletion(Transaction tx) {
    if ( tx.wasCommitted() ) {
        System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
    }
    updates=0;
    creates=0;
    loads=0;
}
}
```

Il y a deux types d'intercepteurs : lié à la `Session` et lié à la `SessionFactory`.

Un intercepteur lié à la `Session` est défini lorsqu'une session est ouverte via l'invocation des méthodes surchargées `SessionFactory.openSession()` acceptant un `Interceptor` (comme argument).

```
Session session = sf.openSession( new AuditInterceptor() );
```

Un intercepteur lié à `SessionFactory` est enregistré avec l'objet `Configuration` avant la construction de la `SessionFactory`. Dans ce cas, les intercepteurs fournis seront appliqués à toutes les sessions ouvertes pour cette `SessionFactory`; ceci est vrai à moins que la session ne soit ouverte en spécifiant l'intercepteur à utiliser. Les intercepteurs liés à la `SessionFactory` doivent être thread safe, en faisant attention à ne pas stocker des états spécifiques de la session puisque plusieurs sessions peuvent utiliser cet intercepteur (potentiellement) de manière concurrente.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

13.2. Système d'événements

Si vous devez réagir à des événements particuliers dans votre couche de persistance, vous pouvez aussi utiliser l'architecture d'événements de Hibernate3. Le système d'événements peut être utilisé en supplément ou en remplacement des interceptors.

Essentiellement toutes les méthodes de l'interface `Session` sont corrélées à un événement. Vous avez un `LoadEvent`, un `FlushEvent`, etc (consultez la DTD du fichier de configuration XML ou le paquetage `org.hibernate.event` pour avoir la liste complète des types d'événement définis). Quand une requête est faite à partir d'une de ces méthodes, la `Session` Hibernate génère un événement approprié et le passe au listener configuré pour ce type. Par défaut, ces listeners implémentent le même traitement dans lequel ces méthodes aboutissent toujours. Cependant, vous êtes libre d'implémenter une version personnalisée d'une de ces interfaces de listener (c'est-à-dire, le `LoadEvent` est traité par l'implémentation de l'interface `LoadEventListener` déclarée), dans quel cas leur implémentation devrait être responsable du traitement des requêtes `load()` faites par la `Session`.

Les listeners devraient effectivement être considérés comme des singletons ; dans le sens où ils sont partagés entre des requêtes, et donc ne devraient pas sauvegarder des états en tant que variables d'instance.

Un listener personnalisé devrait implémenter l'interface appropriée pour l'événement qu'il veut traiter et/ou étendre une des classes de base (ou même l'événement prêt à l'emploi utilisé par Hibernate comme ceux déclarés non-finaux à cette intention). Les listeners personnalisés peuvent être soit inscrits par programmation à travers l'objet `Configuration`, ou spécifiés dans la configuration XML de Hibernate (la configuration déclarative à travers le fichier de propriétés n'est pas supportée). Voici un exemple de listener personnalisé pour l'événement de chargement :

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
        throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

```
}
```

Vous avez aussi besoin d'une entrée de configuration indiquant à Hibernate d'utiliser ce listener en plus du listener par défaut :

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="load">
      <listener class="com.eg.MyLoadListener"/>
      <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
    </event>
  </session-factory>
</hibernate-configuration>
>
```

Vous pouvez aussi l'inscrire par programmation :

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Les listeners inscrits déclarativement ne peuvent pas partager d'instances. Si le même nom de classe est utilisée dans plusieurs éléments `<listener/>`, chaque référence résultera en une instance distincte de cette classe. Si vous avez besoin de la faculté de partager des instances de listener entre plusieurs types de listener, vous devez utiliser l'approche d'inscription par programmation.

Pourquoi implémenter une interface et définir le type spécifique durant la configuration ? Une implémentation de listener pourrait implémenter plusieurs interfaces de listener d'événements. Par ailleurs, le fait de définir le type durant l'inscription, rend l'activation ou la désactivation plus facile au moment de la configuration.

13.3. Sécurité déclarative de Hibernate

Généralement, la sécurité déclarative dans les applications Hibernate est gérée dans la couche de session. Maintenant, Hibernate3 permet à certaines actions d'être approuvées via JACC, et autorisées via JAAS. Cette fonctionnalité optionnelle est construite au dessus de l'architecture d'événements.

D'abord, vous devez configurer les listeners d'événements appropriés pour permettre l'utilisation d'autorisations JAAS.

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
```

```
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Notez que `<listener type="..." class="..."/>` est juste un raccourci pour `<event type="..."><listener class="..."/></event>` quand il y a exactement un listener pour un type d'événement particulier.

Ensuite, toujours dans `hibernate.cfg.xml`, liez les permissions aux rôles :

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions=""/>
```

Les noms de rôle sont les rôles compris par votre fournisseur JAAC.

Traitement par lot

Une approche naïve pour insérer 100 000 lignes dans la base de données en utilisant Hibernate ressemblerait à :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

Ceci devrait s'écrouler avec une `OutOfMemoryException` quelque part aux alentours de la 50 000ème ligne. C'est parce que Hibernate cache toutes les instances de `Customer` nouvellement insérées dans le cache de second niveau. Dans ce chapitre, nous allons vous montrer comment éviter ce problème.

Dans ce chapitre nous montrerons comment éviter ce problème. Mais tout d'abord, si vous faites des traitements par lot, il est absolument indispensable d'activer l'utilisation des lots JDBC, pour obtenir des performances raisonnables. Configurez la taille du lot JDBC à un nombre raisonnable (disons, 10-50) :

```
hibernate.jdbc.batch_size 20
```

Notez que Hibernate désactive, de manière transparente, l'insertion par lot au niveau JDBC si vous utilisez un générateur d'identifiant de type `identity`.

Vous désirez peut-être effectuer ce genre de tâche dans un traitement où l'interaction avec le cache de second niveau est complètement désactivée :

```
hibernate.cache.use_second_level_cache false
```

Toutefois ce n'est pas absolument nécessaire puisque nous pouvons configurer le `CacheMode` de façon à désactiver l'interaction avec le cache de second niveau.

14.1. Insertions en lot

Lorsque vous rendez des nouveaux objets persistants, vous devez régulièrement appeler `flush()` et puis `clear()` sur la session, pour contrôler la taille du cache de premier niveau.

```
Session session = sessionFactory.openSession();
```

```
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

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

14.2. Mise à jour des lots

Pour récupérer et mettre à jour des données les mêmes idées s'appliquent. De plus, vous devez utiliser `scroll()` pour tirer partie des curseurs côté serveur pour les requêtes qui retournent beaucoup de lignes de données.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

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

14.3. L'interface StatelessSession

Alternativement, Hibernate fournit une API orientée commande qui peut être utilisée avec des flux de données vers et en provenance de la base de données sous la forme d'objets détachés. Une `StatelessSession` n'a pas de contexte de persistance associé et ne fournit pas beaucoup de sémantique de cycle de vie de haut niveau. En particulier, une session sans état n'implémente pas de cache de premier niveau et n'interagit pas non plus avec un cache de seconde niveau ou un cache de requêtes. Elle n'implémente pas les transactions ou la vérification sale automatique (automatic dirty checking). Les opérations réalisées avec une session sans état ne sont jamais

répercutées en cascade sur les instances associées. Les collections sont ignorées par une session sans état. Les opérations exécutées via une session sans état outrepassent le modèle d'événements de Hibernate et les intercepteurs. Les sessions sans état sont vulnérables aux effets de réplication des données, ceci est dû au manque de cache de premier niveau. Une session sans état est une abstraction bas niveau, plus proche de la couche JDBC sous-jacente.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

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

Notez que dans le code de l'exemple, les instances de `Customer` retournées par la requête sont immédiatement détachées. Elles ne sont jamais associées à un contexte de persistance.

Les opérations `insert()`, `update()` et `delete()` définies par l'interface `StatelessSession` sont considérées comme des opérations d'accès direct aux lignes de la base de données, ce qui résulte en une exécution immédiate du SQL `INSERT`, `UPDATE` ou `DELETE` respectivement. Ainsi, elles ont des sémantiques très différentes des opérations `save()`, `saveOrUpdate()` et `delete()` définies par l'interface `Session`.

14.4. Opérations de style DML

As already discussed, automatic and transparent object/relational mapping is concerned with the management of the object state. The object state is available in memory. This means that manipulating data directly in the database (using the SQL Data Manipulation Language (DML) the statements: `INSERT`, `UPDATE`, `DELETE`) will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution that is performed through the Hibernate Query Language ([HQL](#)).

La pseudo-syntaxe pour les expressions `UPDATE` et `DELETE` est : `(UPDATE | DELETE) FROM? EntityName (WHERE where_conditions)?`.

Certains points à noter :

- Dans la clause `from`, le mot-clef `FROM` est optionnel
- Il ne peut y avoir qu'une seule entité nommée dans la clause `from` ; elle peut optionnellement avoir un alias. Si le nom de l'entité a un alias, alors n'importe quelle référence de propriété

doit être qualifiée en utilisant un alias ; si le nom de l'entité n'a pas d'alias, il sera illégal pour n'importe quelle référence de propriété d'être qualifiée.

- No *joins*, either implicit or explicit, can be specified in a bulk HQL query. Sub-queries can be used in the where-clause, where the subqueries themselves may contain joins.
- La clause where- est aussi optionnelle.

Par exemple, pour exécuter un HQL UPDATE, utilisez la méthode `Query.executeUpdate()` (la méthode est donnée pour ceux qui connaissent `PreparedStatement.executeUpdate()` de JDBC) :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
    .setString( "newName", newName )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();
```

In keeping with the EJB3 specification, HQL UPDATE statements, by default, do not effect the *version* or the *timestamp* property values for the affected entities. However, you can force Hibernate to reset the version or timestamp property values through the use of a versioned update. This is achieved by adding the VERSIONED keyword after the UPDATE keyword.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
    .setString( "newName", newName )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();
```

Notez que les types personnalisés (`org.hibernate.usertype.UserVersionType`) ne sont pas permis en conjonction avec la déclaration `update versioned`.

Pour exécuter un HQL DELETE, utilisez la même méthode `Query.executeUpdate()` :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
    .setString( "oldName", oldName )
```

```

        .executeUpdate();
tx.commit();
session.close();

```

La valeur du `int` retourné par la méthode `Query.executeUpdate()` indique le nombre d'entités affectées par l'opération. Considérez que cela peut ou non, corréler le nombre de lignes affectées dans la base de données. Une opération HQL pourrait entraîner l'exécution de multiples expressions SQL réelles, pour des classes filles mappées par jointure (join-subclass), par exemple. Le nombre retourné indique le nombre d'entités réelles affectées par l'expression. Si on revient à l'exemple de la classe fille mappée par jointure, un effacement d'une des classes filles peut réellement entraîner des suppressions pas seulement dans la table à laquelle la classe fille est mappée, mais également dans la table "racine" et potentiellement dans les tables des classes filles plus bas dans la hiérarchie d'héritage.

La pseudo-syntaxe pour l'expression `INSERT` est : `INSERT INTO EntityName properties_list select_statement`. Quelques points sont à noter :

- Seule la forme `INSERT INTO ... SELECT ...` est supportée ; pas la forme `INSERT INTO ... VALUES ...`.

La `properties_list` est analogue à la `column specification` dans la déclaration SQL `INSERT`. Pour les entités impliquées dans un héritage mappé, seules les propriétés directement définies à ce niveau de classe donné peuvent être utilisées dans `properties_list`. Les propriétés de la classe mère ne sont pas permises ; et les propriétés des classes filles n'ont pas de sens. En d'autres termes, les expressions `INSERT` sont par nature non polymorphiques.

- `select_statement` peut être n'importe quelle requête de sélection HQL valide, avec l'avertissement que les types de retour doivent correspondre aux types attendus par l'insertion. Actuellement, cela est vérifié durant la compilation de la requête plutôt que de reléguer la vérification à la base de données. Notez cependant que cela pourrait poser des problèmes entre les `Types` de Hibernate qui sont *équivalents* contrairement à *égaux*. Cela pourrait poser des problèmes avec des disparités entre une propriété définie comme un `org.hibernate.type.DateType` et une propriété définie comme un `org.hibernate.type.TimestampType`, bien que la base de données ne fasse pas de distinction ou ne soit pas capable de gérer la conversion.
- Pour la propriété `id`, l'expression d'insertion vous donne deux options. Vous pouvez soit spécifier explicitement la propriété `id` dans `properties_list` (auquel cas sa valeur est extraite de l'expression de sélection correspondante), soit l'omettre de `properties_list` (auquel cas une valeur générée est utilisée). Cette dernière option est seulement disponible si vous utilisez le générateur d'identifiant qui opère dans la base de données ; tenter d'utiliser cette option avec n'importe quel type de générateur "en mémoire" causera une exception durant l'analyse. Notez que pour les buts de cette discussion, les générateurs "en base" sont considérés comme `org.hibernate.id.SequenceGenerator` (et ses classes filles) et n'importe quelles implémentations de `org.hibernate.id.PostInsertIdentifierGenerator`. L'exception la plus notable ici est `org.hibernate.id.TableHiLoGenerator`, qui ne peut pas être utilisée parce qu'elle ne propose pas de moyen d'obtenir ses valeurs par un `select`.

- Pour des propriétés mappées comme `version` ou `timestamp`, l'expression d'insertion vous donne deux options. Vous pouvez soit spécifier la propriété dans `properties_list` (auquel cas sa valeur est extraite des expressions `select` correspondantes), soit l'omettre de `properties_list` (auquel cas la `seed value` définie par le `org.hibernate.type.VersionType` est utilisée).

Un exemple d'exécution d'une expression HQL `INSERT` :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer
c where ...";
int createdEntities = s.createQuery( hqlInsert )
    .executeUpdate();
tx.commit();
session.close();
```

HQL : langage d'interrogation d'Hibernate

Hibernate fournit un langage d'interrogation extrêmement puissant qui ressemble (et c'est voulu) au SQL. Mais ne soyez pas dupe de la syntaxe ; HQL est totalement orienté objet, cernant des notions comme l'héritage, le polymorphisme et les associations.

15.1. Sensibilité à la casse

Les requêtes sont insensibles à la casse, à l'exception des noms de classes Java et des propriétés. Ainsi, `SeLeCT` est identique à `seLEct` et à `SELECT` mais `net.sf.hibernate.eg.FOO` n'est pas identique à `net.sf.hibernate.eg.foo` et `foo.barSet` n'est pas identique à `foo.BARSET`.

Ce guide utilise les mots clés HQL en minuscules. Certains utilisateurs trouvent les requêtes écrites avec les mots clés en majuscules plus lisibles, mais nous trouvons cette convention pénible lorsqu'elle est lue dans du code Java.

15.2. La clause `from`

La requête Hibernate la plus simple est de la forme :

```
from eg.Cat
```

Retourne toutes les instances de la classe `eg.Cat`. Nous n'avons pas besoin de qualifier le nom de la classe, puisque `auto-import` est la valeur par défaut. Donc nous écrivons presque toujours :

```
from Cat
```

Pour pouvoir nous référer à `Cat` dans des autres parties de la requête, vous aurez besoin d'y assigner un *alias*. Ainsi :

```
from Cat as cat
```

Cette requête assigne l'alias `cat` à l'instance `Cat`, nous pouvons donc utiliser cet alias ailleurs dans la requête. Le mot clé `as` est optionnel. Nous aurions pu écrire :

```
from Cat cat
```

Plusieurs classes peuvent apparaître, ce qui conduira à un produit cartésien (encore appelé jointures croisées).

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

C'est une bonne pratique que de nommer les alias dans les requêtes en utilisant l'initiale en miniscule, ce qui correspond aux standards de nommage Java pour les variables locales (par ex. `domesticCat`).

15.3. Associations et jointures

On peut aussi assigner des alias à des entités associées, ou même aux éléments d'une collection de valeurs, en utilisant un `join` (jointure). Par exemple :

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

Les types de jointures supportées sont empruntées de ANSI SQL :

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (jointure ouverte totalement - généralement inutile)

Les constructions des jointures `inner join`, `left outer join` **et** `right outer join` peuvent être abrégées.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Nous pouvons soumettre des conditions de jointure supplémentaires en utilisant le mot-clef HQL `with`.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight
> 10.0
```

A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See [Section 20.1, « Stratégies de chargement »](#) for more information.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Une jointure "fetch" (rapportée) n'a généralement pas besoin de se voir assigner un alias puisque les objets associés ne doivent pas être utilisés dans la clause `where` ou toute autre clause. Notez aussi que les objets associés ne sont pas retournés directement dans le résultat de la requête mais l'on peut y accéder via l'objet parent. La seule raison pour laquelle nous pourrions avoir besoin d'un alias est si nous récupérons récursivement une collection supplémentaire :

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

Notez que la construction de `fetch` ne peut pas être utilisée dans les requêtes appelées par `scroll()` ou `iterate()`. De même `Fetch` ne devrait pas être utilisé avec `setMaxResults()` ou `setFirstResult()`, ces opérations étant basées sur le nombre de résultats contenant généralement des doublons dès que des collections sont chargées agressivement, par conséquent le nombre de lignes est imprévisible. `Fetch` ne peut pas non plus être utilisé avec une condition `with` ad hoc. Il est possible de créer un produit cartésien par jointure en récupérant plus d'une collection dans une requête, donc faites attention dans ce cas. Récupérer par jointure de multiples collections donne aussi parfois des résultats inattendus pour des mappages de sac, donc soyez prudent lorsque vous formulez vos requêtes dans de tels cas. Finalement, notez que `full join fetch` et `right join fetch` ne sont pas utiles en général.

Si vous utilisez un chargement retardé pour les propriétés (avec une instrumentation par bytecode), il est possible de forcer Hibernate à récupérer les propriétés non encore chargées immédiatement (dans la première requête) en utilisant `fetch all properties`.

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

15.4. Formes de syntaxes pour les jointures

HQL supporte deux formes pour joindre les associations : `implicit` et `explicit`.

Les requêtes présentes dans la section précédente utilisent la forme `explicit` où le mot clé `join` est explicitement utilisé dans la clause `from`. C'est la forme recommandée.

La forme `implicit` n'utilise pas le mot clé `join`. En revanche, les associations sont "déréférencées" en utilisant la notation. Ces jointures `implicit` peuvent apparaître dans toutes les clauses HQL. Les jointures `implicit` résultent en des jointures internes dans le SQL généré.

```
from Cat as cat where cat.mate.name like '%s%'
```

15.5. Faire référence à la propriété identifiant

Il y a en général deux façons de faire référence à une propriété d'identifiant d'une entité :

- La propriété particulière (minuscule) `id` peut être utilisée pour référencer la propriété d'identifiant d'une entité *du moment que l'entité ne définit pas une propriété de non-identifiant appelée `id`*.
- Si l'entité définit une propriété d'identifiant nommée, vous pouvez utiliser ce nom de propriété.

Les références aux propriétés d'identifiant composites suivent les mêmes règles de nommage. Si l'entité a une propriété de non-identifiant appelée `id`, la propriété d'identifiant composite ne peut être référencée que par son nom défini ; sinon la propriété spéciale `id` peut être utilisée pour référencer la propriété d'identifiant.



Important

Note : cela a changé de façon significative depuis la version 3.2.2. Dans les versions précédentes, `id` référait *toujours* à la propriété identifiant quel que soit son nom réel. Une des conséquences de cette décision fut que les propriétés de non-identifiant appelées `id` ne pouvaient jamais être référencées dans les requêtes Hibernate.

15.6. La clause select

La clause `select` sélectionne les objets et propriétés qui doivent être retournés dans le résultat de la requête. Soit :

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

La requête recherchera les `mate` s liés aux `Cat` s. Vous pouvez exprimer cette requête de manière plus compacte :

```
select cat.mate from Cat cat
```

Les requêtes peuvent retourner des propriétés de n'importe quel type de valeur, même celles de type composant :

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Les requêtes peuvent retourner de multiples objets et/ou propriétés sous la forme d'un tableau du type `Object[]` :

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

Ou sous la forme d'une `List` :

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

Ou bien - à condition que la classe `Family` possède le constructeur approprié - en tant qu'objet `typesafe Java` :

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Vous pouvez assigner des alias aux expressions sélectionnées en utilisant `as` :

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

C'est surtout utile lorsque c'est utilisé avec `select new map` :

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Cette requête retourne une `Map` à partir des alias vers les valeurs sélectionnées.

15.7. Fonctions d'agrégation

Les requêtes HQL peuvent aussi retourner les résultats de fonctions d'agrégation sur les propriétés :

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

Les fonctions d'agrégation supportées sont :

- `avg(...)`, `sum(...)`, `min(...)`, `max(...)`
- `count(*)`
- `count(...)`, `count(distinct ...)`, `count(all...)`

Vous pouvez utiliser des opérateurs arithmétiques, la concaténation, et des fonctions SQL reconnues dans la clause `select` :

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

Les mots clé `distinct` et `all` peuvent être utilisés et ont la même sémantique qu'en SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

15.8. Requêtes polymorphiques

Une requête comme :

```
from Cat as cat
```

retourne non seulement les instances de `Cat`, mais aussi celles des sous classes comme `DomesticCat`. Les requêtes Hibernate peuvent nommer n'importe quelle classe ou interface Java dans la clause `from`. La requête retournera les instances de toutes les classes persistantes qui étendent cette classe ou implémente cette interface. La requête suivante retournera tous les objets persistants :

```
from java.lang.Object o
```

L'interface `Named` peut être implémentée par plusieurs classes persistantes :

```
from Named n, Named m where n.name = m.name
```

Notez que ces deux dernières requêtes nécessitent plus d'un SQL `SELECT`. Ce qui signifie que la clause `order by` ne trie pas correctement la totalité des résultats (cela signifie aussi que vous ne pouvez exécuter ces requêtes en appelant `Query.scroll()`).

15.9. La clause where

La clause `where` vous permet de réduire la liste des instances retournées. Si aucun alias n'existe, vous pouvez vous référer aux propriétés par leur nom :

```
from Cat where name='Fritz'
```

S'il y a un alias, utilisez un nom de propriété qualifié :

```
from Cat as cat where cat.name='Fritz'
```

Retourne les instances de `Cat` appelé 'Fritz'.

La requête suivante :

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

retournera les instances de `Foo` pour lesquelles il existe une instance de `bar` avec la propriété `date` égale à la propriété `startDate` de `Foo`. Les expressions de chemin composées rendent la clause `where` extrêmement puissante. Soit :

```
from Cat cat where cat.mate.name is not null
```

Cette requête se traduit en une requête SQL par une jointure interne de table. Si vous souhaitez écrire quelque chose comme :

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

vous finiriez avec une requête qui nécessiterait quatre jointures de table en SQL.

L'opérateur `=` peut être utilisé pour comparer aussi bien des propriétés que des instances :

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

The special property (lowercase) `id` can be used to reference the unique identifier of an object. See [Section 15.5, « Faire référence à la propriété identifiant »](#) for more information.

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

La seconde requête est particulièrement efficace. Aucune jointure n'est nécessaire !

Les propriétés d'identifiants composites peuvent aussi être utilisées. Supposez que `Person` ait un identifiant composite composé de `country` et `medicareNumber`.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Une fois de plus, la seconde requête ne nécessite pas de jointure de table.

See [Section 15.5, « Faire référence à la propriété identifiant »](#) for more information regarding referencing identifier properties)

De même, la propriété spéciale `class` accède à la valeur discriminante d'une instance dans le cas d'une persistance polymorphique. Le nom d'une classe Java incorporée dans la clause `where` sera traduite par sa valeur discriminante.

```
from Cat cat where cat.class = DomesticCat
```

You can also use components or composite user types, or properties of said component types. See [Section 15.17, « Composants »](#) for more information.

Un type "any" possède les propriétés particulières `id` et `class`, qui nous permettent d'exprimer une jointure de la manière suivante (là où `AuditLog.item` est une propriété mappée avec `<any>`) :

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

Dans la requête précédente, notez que `log.item.class` et `payment.class` feraient référence à des valeurs de colonnes de la base de données complètement différentes.

15.10. Expressions

Les expressions permises dans la clause `where` incluent :

- opérateurs mathématiques : `+`, `-`, `*`, `/`
- opérateurs de comparaison binaire : `=`, `>=`, `<=`, `<>`, `!=`, `like`
- opérations logiques : `and`, `or`, `not`
- Parenthèses (`)`, indiquant un regroupement
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` et `not member of`
- Cas simple `case ... when ... then ... else ... end`, et cas "searched", `case when ... then ... else ... end`

- concaténation de chaîne de caractères `... || ...` ou `concat(..., ...)`
- `current_date()`, `current_time()`, and `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`,
- N'importe quelle fonction ou opérateur défini par EJB-QL 3.0 : `substring()`, `trim()`, `lower()`, `upper()`, `length()`, `locate()`, `abs()`, `sqrt()`, `bit_length()`, `mod()`
- `coalesce()` et `nullif()`
- `str()` pour convertir des valeurs numériques ou temporelles vers une chaîne de caractères lisible
- `cast(... as ...)`, où le second argument est le nom d'un type Hibernate, et `extract(... from ...)` si le `cast()` ANSI et `extract()` sont supportés par la base de données sous-jacente
- La fonction HQL `index()`, qui s'applique aux alias d'une collection indexée jointe
- Les fonctions HQL qui prennent des expressions de chemin représentant des collections : `size()`, `minelement()`, `maxelement()`, `minindex()`, `maxindex()`, ainsi que les fonctions particulières `elements()` et `indices` qui peuvent être quantifiées en utilisant `some`, `all`, `exists`, `any`, `in`.
- N'importe quelle fonction scalaire SQL supportée par la base de données comme `sign()`, `trunc()`, `rtrim()`, et `sin()`
- Les paramètres de position de JDBC ?
- paramètres nommés `:name`, `:start_date`, et `:x1`
- SQL textuel `'foo'`, `69`, `6.66E+2`, `'1970-01-01 10:00:01.0'`
- Constantes Java `public static final` `Color.TABBY`

`in` et `between` peuvent être utilisés comme suit :

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

Les formes négatives peuvent être écrites ainsi :

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

De même, `is null` et `is not null` peuvent être utilisés pour tester les valeurs nulles.

Les booléens peuvent être facilement utilisés en déclarant les substitutions de requêtes HQL dans la configuration Hibernate :

```
<property name="hibernate.query.substitutions"
```

```
>true 1, false 0</property>
>
```

Ce qui remplacera les mots clés `true` et `false` par 1 et 0 dans la traduction SQL du HQL suivant :

```
from Cat cat where cat.alive = true
```

Vous pouvez tester la taille d'une collection par la propriété particulière `size`, ou la fonction spéciale `size()`.

```
from Cat cat where cat.kittens.size
> 0
```

```
from Cat cat where size(cat.kittens)
> 0
```

Pour les collections indexées, vous pouvez faire référence aux indices minimum et maximum en utilisant les fonctions `minindex` et `maxindex`. De manière similaire, vous pouvez faire référence aux éléments minimum et maximum d'une collection de type basique en utilisant les fonctions `minelement` et `maxelement`. Par exemple :

```
from Calendar cal where maxelement(cal.holidays)
> current_date
```

```
from Order order where maxindex(order.items)
> 100
```

```
from Order order where minelement(order.items)
> 10000
```

Les fonctions SQL `any`, `some`, `all`, `exists`, `in` sont supportées quand l'élément ou l'ensemble des indexes d'une collection (les fonctions `elements` et `indices`) ou le résultat d'une sous requête sont passés (voir ci dessous) :

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3
> all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

Notez que l'écriture de `- size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement` - peut seulement être utilisée dans la clause `where` dans `Hibernate3`.

Les éléments de collections indexées (arrays, lists, maps) peuvent être référencés via `index` dans une clause `where` seulement :

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

L'expression entre `[]` peut même être une expression arithmétique :

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL propose aussi une fonction `index()` interne, pour les éléments d'une association un-à-plusieurs ou d'une collection de valeurs.


```
select item, index(item) from Order order
      join order.items item
where index(item) < 5
```

Les fonctions SQL scalaires supportées par la base de données utilisée peuvent être utilisées :

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

Si vous n'êtes pas encore convaincu par tout cela, imaginez la taille et l'illisibilité qui caractériseraient la requête suivante en SQL :

```
select cust
from Product prod,
      Store store
      inner join store.customers cust
where prod.name = 'widget'
      and store.location.name in ( 'Melbourne', 'Sydney' )
      and prod = all elements(cust.currentOrder.lineItems)
```

Un indice : cela donne quelque chose comme

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
      stores store,
      locations loc,
      store_customers sc,
      product prod
WHERE prod.name = 'widget'
      AND store.loc_id = loc.id
      AND loc.name IN ( 'Melbourne', 'Sydney' )
      AND sc.store_id = store.id
      AND sc.cust_id = cust.id
      AND prod.id = ALL(
      SELECT item.prod_id
      FROM line_items item, orders o
      WHERE item.order_id = o.id
            AND cust.current_order = o.id
      )
```

15.11. La clause order by

La liste retournée par la requête peut être triée par n'importe quelle propriété de la classe ou des composants retournés :

```
from DomesticCat cat
```

```
order by cat.name asc, cat.weight desc, cat.birthdate
```

Le mot optionnel `asc` ou `desc` indique respectivement si le tri doit être croissant ou décroissant.

15.12. La clause `group by`

Si la requête retourne des valeurs agrégées, celles-ci peuvent être groupées par propriété d'une classe retournée ou par des composants :

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

Une clause `having` est aussi permise.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Les fonctions SQL et les fonctions d'agrégat sont permises dans les clauses `having` et `order by`, si elles sont prises en charge par la base de données sous-jacente (ce que ne fait pas MySQL par exemple).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight)
> 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Notez que ni la clause `group by` ni la clause `order by` ne peuvent contenir d'expressions arithmétiques. Notez aussi qu'Hibernate ne développe pas actuellement une entité faisant partie du regroupement, donc vous ne pouvez pas écrire `group by cat` si toutes les propriétés de `cat` sont non-agrégées. Vous devez lister toutes les propriétés non-agrégées explicitement.

15.13. Sous-requêtes

Pour les bases de données supportant les sous-selects, Hibernate supporte les sous requêtes dans les requêtes. Une sous-requête doit être entre parenthèses (souvent pour un appel à une fonction d'agrégation SQL). Même les sous-requêtes corrélées (celles qui font référence à un alias de la requête principale) sont supportées.

```
from Cat as fatcat
where fatcat.weight
> (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Notez que les sous-requêtes HQL peuvent survenir uniquement dans les clauses select ou where.

Note that subqueries can also utilize `row value constructor` syntax. See [Section 15.18](#), « *Syntaxe des constructeurs de valeur de ligne* » for more information.

15.14. Exemples HQL

Les requêtes Hibernate peuvent être relativement puissantes et complexes. En fait, la puissance du langage d'interrogation est l'un des arguments principaux de vente de Hibernate. Voici quelques exemples très similaires aux requêtes que nous avons utilisées lors d'un récent projet.

Notez que la plupart des requêtes que vous écrirez seront plus simples que les exemples qui suivent.

La requête suivante retourne l'id de commande, le nombre d'articles et la valeur totale de la commande pour toutes les commandes non payées d'un client particulier pour une valeur totale minimum donnée, ces résultats étant triés par la valeur totale. La requête SQL générée sur les tables ORDER, ORDER_LINE, PRODUCT, CATALOG et PRICE est composée de quatre jointures internes ainsi que d'un sous-select (non corrélé).

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate
>= all (
    select cat.effectiveDate
    from Catalog as cat
    where cat.effectiveDate < sysdate
)
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

Quel monstre ! En principe, dans des situations réelles, nous n'approuvons pas les sous-requêtes, notre requête ressemblait donc plutôt à ce qui suit :

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

La requête suivante compte le nombre de paiements pour chaque statut, en excluant tout paiement dans le statut AWAITING_APPROVAL où le changement de statut le plus récent à été fait

par l'utilisateur courant. En SQL, cette requête effectue deux jointures internes et un sous-select corrélié sur les tables `PAYMENT`, `PAYMENT_STATUS` et `PAYMENT_STATUS_CHANGE`.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <
> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

Si nous avons mappé la collection `statusChanges` comme une liste, au lieu d'un ensemble, la requête aurait été plus facile à écrire.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <
> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

La prochaine requête utilise la fonction de serveur MS SQL `isNull()` pour retourner tous les comptes et paiements impayés pour l'organisation à laquelle l'utilisateur courant appartient. Elle est traduite en SQL par trois jointures internes, une jointure externe ainsi qu'un sous-select sur les tables `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` et `ORG_USER`.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

Pour certaines bases de données, nous devons éliminer le sous-select (corrélié).

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

15.15. Nombreuses mises à jour et suppressions

HQL now supports update, delete and insert ... select ... statements. See [Section 14.4](#), « *Opérations de style DML* » for more information.

15.16. Trucs & Astuces

Vous pouvez compter le nombre de résultats d'une requête sans les retourner :

```
( (Integer) session.createQuery("select count(*) from ...").iterate().next() ).intValue()
```

Pour trier les résultats par la taille d'une collection, utilisez la requête suivante :

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

Si votre base de données supporte les sous-selects, vous pouvez placer des conditions sur la taille de la sélection dans la clause where de votre requête :

```
from User usr where size(usr.messages)
>= 1
```

Si votre base de données ne supporte pas les sous-selects, utilisez la requête suivante :

```
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg)
>= 1
```

Cette solution ne peut pas retourner un `User` avec zéro message à cause de la jointure interne, la forme suivante peut donc être utile :

```
select usr.id, usr.name
from User as usr
      left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Les propriétés d'un `JavaBean` peuvent être injectées dans les paramètres nommés d'une requête :

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

Les collections sont paginables via l'utilisation de l'interface `Query` avec un filtre :

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Les éléments d'une collection peuvent être triés ou groupés en utilisant un filtre de requête :

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Vous pouvez récupérer la taille d'une collection sans l'initialiser :

```
( (Integer) session.createQuery("select count(*) from ...").iterate().next() ).intValue();
```

15.17. Composants

Les composants peuvent être utilisés dans presque tous les cas comme les types de valeur dans les requêtes HQL. Ils peuvent apparaître dans la clause `select` comme ce qui suit :

```
select p.name from Person p
```

```
select p.name.first from Person p
```

où la propriété de nom de `Person` est un composant. Des composants peuvent aussi être utilisés dans la clause `where` :

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

Des composants peuvent être utilisés dans la clause `order by` :

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

Another common use of components is in [row value constructors](#).

15.18. Syntaxe des constructeurs de valeur de ligne

HQL supporte l'utilisation de la syntaxe `row value constructor` SQL ANSI (aussi appelée syntaxe `tuple`), bien que la base de données sous-jacente ne supporte pas nécessairement cette notion. Là, nous faisons généralement référence à des comparaisons multi-valuées, typiquement associées à des composants. Considérez une entité `Person` qui définit un composant de nom :

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

Voici une syntaxe valide, bien que quelque peu fastidieuse. Pour la rendre plus concise, utilisez la syntaxe `row value constructor` :

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

Il est utile de spécifier cela dans la clause `select` :

```
select p.name from Person p
```

Alternativement, utiliser la syntaxe `row value constructor` peut être avantageux quand vous utilisez des sous-requêtes nécessitant une comparaison avec des valeurs multiples :


```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

Si vous décidez d'utiliser cette syntaxe, il vous faudra prendre en considération le fait que la requête sera dépendante de la commande des sous-propriétés du composant dans les métadonnées.

Requêtes par critères

Hibernate offre une API d'interrogation par critères intuitive et extensible.

16.1. Créer une instance de `Criteria`

L'interface `net.sf.hibernate.Criteria` représente une requête sur une classe persistante donnée. La `Session` fournit les instances de `Criteria`.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

16.2. Restriction du résultat

Un critère de recherche (criterion) individuel est une instance de l'interface `org.hibernate.criterion.Criterion`. La classe `org.hibernate.criterion.Restrictions` définit des méthodes de fabrique pour obtenir des types de `Criterion` intégrés.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Les restrictions peuvent être groupées de manière logique.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    )
    .list();
```

Il y a un grand choix de types de critères intégrés (sous classes de `Restriction`), dont un est particulièrement utile puisqu'il vous permet de spécifier directement SQL.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)", "Fritz
%", Hibernate.STRING) )
    .list();
```

La zone `{alias}` sera remplacée par l'alias de colonne de l'entité que l'on souhaite interroger.

Une autre approche pour obtenir un critère est de le récupérer d'une instance de `Property`. Vous pouvez créer une `Property` en appelant `Property.forName()`.

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

16.3. Trier les résultats

Vous pouvez trier les résultats en utilisant `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

16.4. Associations

En naviguant les associations qui utilisent `createCriteria()`, vous pouvez spécifier des contraintes associées à des entités :

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

Notez que la seconde `createCriteria()` retourne une nouvelle instance de `Criteria`, qui se rapporte aux éléments de la collection `kittens`.

La forme alternative suivante est utile dans certains cas :

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` ne crée pas de nouvelle instance de `Criteria`.)

Notez que les collections `kittens` contenues dans les instances de `Cat` retournées par les deux précédentes requêtes ne sont *pas* pré-filtrées par les critères ! Si vous souhaitez récupérer uniquement les `kittens` correspondant aux critères, vous devez utiliser `ResultTransformer`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

Additionally you may manipulate the result set using a left outer join:

```
List cats = session.createCriteria( Cat.class )
    .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name",
"good%") )
    .addOrder(Order.asc("mt.age"))
```

```
.list();
```

This will return all of the `Cats` with a mate whose name starts with "good" ordered by their mate's age, and all cats who do not have a mate. This is useful when there is a need to order or limit in the database prior to returning complex/large result sets, and removes many instances where multiple queries would have to be performed and the results unioned by java in memory.

Without this feature, first all of the cats without a mate would need to be loaded in one query.

A second query would need to retrieve the cats with mates whose name started with "good" sorted by the mates age.

Thirdly, in memory; the lists would need to be joined manually.

16.5. Peuplement d'associations de manière dynamique

Vous pouvez spécifier, au moment de l'exécution, le peuplement d'une association en utilisant `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join. See [Section 20.1, « Stratégies de chargement »](#) for more information.

16.6. Requêtes par l'exemple

La classe `org.hibernate.criterion.Example` vous permet de construire un critère de requête à partir d'une instance d'objet donnée.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Les propriétés de type version, identifiant et association sont ignorées. Par défaut, les valeurs null sont exclues.

Vous pouvez ajuster la stratégie d'utilisation de valeurs de l'`Example`.

```

Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();

```

Vous pouvez utiliser les "exemples" pour des critères sur des objets associés.

```

List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();

```

16.7. Projections, agrégation et regroupement

La classe `org.hibernate.criterion.Projections` est une fabrique d'instances de `Projection`. Nous appliquons une projection sur une requête en appelant `setProjection()`.

```

List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();

```

```

List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();

```

Il n'y a pas besoin de "group by" explicite dans une requête par critère. Certains types de projection sont définis pour être des *projections de regroupement*, qui apparaissent aussi dans la clause SQL `group by`.

Un alias peut optionnellement être assigné à une projection, ainsi la valeur projetée peut être référencée dans des restrictions ou des tris. À cet effet, voici deux procédés différents :

```

List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )

```

```
.addOrder( Order.asc("color") )  
.list();
```

```
List results = session.createCriteria(Cat.class)  
.setProjection( Projections.groupProperty("color").as("color") )  
.addOrder( Order.asc("color") )  
.list();
```

Les méthodes `alias()` et `as()` enveloppent simplement une instance de projection dans une autre instance (aliasée) de `Projection`. Pour un raccourci, vous pouvez assigner un alias lorsque vous ajoutez la projection à une liste de projections :

```
List results = session.createCriteria(Cat.class)  
.setProjection( Projections.projectionList()  
.add( Projections.rowCount(), "catCountByColor" )  
.add( Projections.avg("weight"), "avgWeight" )  
.add( Projections.max("weight"), "maxWeight" )  
.add( Projections.groupProperty("color"), "color" )  
)  
.addOrder( Order.desc("catCountByColor") )  
.addOrder( Order.desc("avgWeight") )  
.list();
```

```
List results = session.createCriteria(Domestic.class, "cat")  
.createAlias("kittens", "kit")  
.setProjection( Projections.projectionList()  
.add( Projections.property("cat.name"), "catName" )  
.add( Projections.property("kit.name"), "kitName" )  
)  
.addOrder( Order.asc("catName") )  
.addOrder( Order.asc("kitName") )  
.list();
```

Vous pouvez aussi utiliser `Property.forName()` pour formuler des projections :

```
List results = session.createCriteria(Cat.class)  
.setProjection( Property.forName("name") )  
.add( Property.forName("color").eq(Color.BLACK) )  
.list();
```

```
List results = session.createCriteria(Cat.class)  
.setProjection( Projections.projectionList()  
.add( Projections.rowCount().as("catCountByColor") )  
.add( Property.forName("weight").avg().as("avgWeight") )  
.add( Property.forName("weight").max().as("maxWeight") )  
.add( Property.forName("color").group().as("color") )  
)
```



```

)
.addOrder( Order.desc( "catCountByColor" ) )
.addOrder( Order.desc( "avgWeight" ) )
.list();

```

16.8. Requêtes et sous-requêtes détachées

La classe `DetachedCriteria` vous laisse créer une requête en dehors de la portée de la session, et puis l'exécuter plus tard en utilisant une `Session` arbitraire.

```

DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName( "sex" ).eq( 'F' ) );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();

```

Les `DetachedCriteria` peuvent aussi être utilisés pour exprimer une sous-requête. Des instances de critère impliquant des sous-requêtes peuvent être obtenues via `Subqueries` ou `Property`.

```

DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName( "weight" ).avg() );
session.createCriteria(Cat.class)
    .add( Property.forName( "weight" ).gt( avgWeight ) )
    .list();

```

```

DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName( "weight" ) );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll( "weight", weights ) )
    .list();

```

Des sous-requêtes corrélées sont également possibles :

```

DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName( "weight" ).avg() )
    .add( Property.forName( "cat2.sex" ).eqProperty( "cat.sex" ) );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName( "weight" ).gt( avgWeightForSex ) )
    .list();

```

16.9. Requêtes par identifiant naturel

Pour la plupart des requêtes, incluant les requêtes par critère, le cache de requêtes n'est pas très efficace, parce que l'invalidation du cache de requêtes arrive trop souvent. Cependant, il y existe une requête spéciale où l'on peut optimiser l'algorithme d'invalidation du cache : les recherches par une clef naturelle constante. Dans certaines applications, ce genre de requête se produit fréquemment. L'API des critères fournit une disposition spéciale pour ce cas d'utilisation.

D'abord, vous devrez mapper la clé naturelle de votre entité en utilisant `<natural-id>`, et activer l'utilisation du cache de second niveau.

```
<class name="User">
  <cache usage="read-write"/>
  <id name="id">
    <generator class="increment"/>
  </id>
  <natural-id>
    <property name="name"/>
    <property name="org"/>
  </natural-id>
  <property name="password"/>
</class>
>
```

Cette fonctionnalité n'est pas prévue pour l'utilisation avec des entités avec des clés naturelles *mutables*.

Une fois que vous aurez activé le cache de requête d'Hibernate, `Restrictions.naturalId()` vous permettra de rendre l'utilisation de l'algorithme de cache plus efficace.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    )
    .setCacheable(true)
    .uniqueResult();
```

SQL natif

Vous pouvez aussi écrire vos requêtes dans le dialecte SQL natif de votre base de données. Ceci est utile si vous souhaitez utiliser les fonctionnalités spécifiques de votre base de données comme le mot clé `CONNECT` d'Oracle. Cette fonctionnalité offre par ailleurs un moyen de migration plus propre et doux d'une application basée directement sur SQL/JDBC vers Hibernate.

Hibernate3 vous permet de spécifier du SQL écrit à la main (y compris les procédures stockées) pour toutes les opérations de création, mise à jour, suppression et chargement.

17.1. Utiliser une requête SQLQuery

L'exécution des requêtes en SQL natif est contrôlée par l'interface `SQLQuery`, qui est obtenue en appelant `Session.createQuery()`. Ce qui suit décrit comment utiliser cette API pour les requêtes.

17.1.1. Requêtes scalaires

La requête SQL la plus basique permet de récupérer une liste de (valeurs) scalaires.

```
sess.createQuery("SELECT * FROM CATS").list();
sess.createQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

Ces deux requêtes retourneront un tableau d'objets (`Object[]`) avec les valeurs scalaires de chacune des colonnes de la table `CATS`. Hibernate utilisera le `ResultSetMetadata` pour déduire l'ordre final et le type des valeurs scalaires retournées.

Pour éviter l'overhead lié à `ResultSetMetadata` ou simplement pour être plus explicite dans ce qui est retourné, vous pouvez utiliser `addScalar()`.

```
sess.createQuery("SELECT * FROM CATS")
    .addScalar("ID", Hibernate.LONG)
    .addScalar("NAME", Hibernate.STRING)
    .addScalar("BIRTHDATE", Hibernate.DATE)
```

Cette requête spécifie :

- la chaîne de requêtes SQL
- les colonnes et les types retournés

Cela retournera toujours un tableau d'objets, mais sans utiliser le `ResultSetMetadata`. Il récupérera à la place explicitement les colonnes `ID`, `NAME` et `BIRTHDATE` comme étant

respectivement de type Long, String et Short, depuis l'ensemble de résultats sous-jacent. Cela signifie aussi que seules ces trois colonnes seront retournées même si la requête utilise `*` et pourrait retourner plus que les trois colonnes listées.

Il est possible de ne pas définir l'information sur le type pour toutes ou une partie des scalaires.

```
sess.createSQLQuery("SELECT * FROM CATS")
    .addScalar("ID", Hibernate.LONG)
    .addScalar("NAME")
    .addScalar("BIRTHDATE")
```

Il s'agit essentiellement de la même requête que précédemment, mais le `ResultSetMetaData` est utilisé pour décider des types de NAME et BIRTHDATE alors que le type de ID est explicitement spécifié.

Les `java.sql.Types` retournés par le `ResultSetMetaData` sont mappés aux types Hibernate via le `Dialect`. Si un type spécifique n'est pas mappé ou est mappé à un type non souhaité, il est possible de le personnaliser en invoquant `registerHibernateType` dans le `Dialect`.

17.1.2. Requêtes d'entités

Les requêtes précédentes ne retournaient que des valeurs scalaires, en ne retournant que les valeurs brutes de l'ensemble de résultats. Ce qui suit montre comment récupérer des entités depuis une requête native SQL, grâce à `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

Cette requête spécifie :

- la chaîne de requêtes SQL
- L'entité retournée par la requête

Avec `Cat` mappé comme classe avec les colonnes ID, NAME et BIRTHDATE, les requêtes précédentes retournent toutes deux, une liste où chaque élément est une entité `Cat`.

Si l'entité est mappée avec un `many-to-one` vers une autre entité, il est requis de retourner aussi cette entité en exécutant la requête native, sinon une erreur "column not found" spécifique à la base de données sera soulevée. Les colonnes additionnelles seront automatiquement retournées en utilisant la notation `*`, mais nous préférons être explicites comme dans l'exemple suivant avec le `many-to-one` vers `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

Ceci permet à `cat.getDog()` de fonctionner normalement.

17.1.3. Gérer les associations et collections

Il est possible de charger agressivement `Dog` pour éviter le chargement de proxies c'est-à-dire un aller-retour supplémentaire vers la base de données. Ceci est effectué via la méthode `addJoin()`, qui vous permet de joindre une association ou collection.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d
WHERE c.DOG_ID = d.D_ID")
.addEntity("cat", Cat.class)
.addJoin("cat.dog");
```

Dans cet exemple, les `Cat` retournés auront leur propriété `dog` entièrement initialisée sans aucun aller-retour supplémentaire vers la base de données. Notez que nous avons ajouté un alias ("cat") pour être capable de spécifier le chemin de la propriété cible de la jointure. Il est possible de faire la même jointure agressive pour les collections, par ex. si le `Cat` a un un-à-plusieurs vers `Dog`.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
c.ID = d.CAT_ID")
.addEntity("cat", Cat.class)
.addJoin("cat.dogs");
```

À ce stade, nous arrivons aux limites de ce qui est possible avec les requêtes natives sans modifier les requêtes SQL pour les rendre utilisables par Hibernate; les problèmes surviennent lorsque nous essayons de retourner des entités du même type ou lorsque les alias/colonnes par défaut ne sont plus suffisants.

17.1.4. Retour d'entités multiples

Jusqu'à présent, les colonnes de l'ensemble de résultats sont supposées être les mêmes que les noms de colonnes spécifiés dans les documents de mapping. Ceci peut être problématique pour les requêtes SQL qui effectuent de multiples jointures vers différentes tables, puisque les mêmes colonnes peuvent apparaître dans plus d'une table.

L'injection d'alias de colonne est requise pour la requête suivante (qui risque de ne pas fonctionner) :

```
sess.createSQLQuery("SELECT c.*, m.* FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
.addEntity("cat", Cat.class)
.addEntity("mother", Cat.class)
```

Le but de cette requête est de retourner deux instances de `Cat` par ligne, un chat et sa mère. Cela échouera puisqu'il y a conflit de noms puisqu'ils sont mappés au même nom de colonne et que sur certaines base de données, les alias de colonnes retournés seront plutôt de la forme

"c.ID", "c.NAME", etc. qui ne sont pas égaux aux colonnes spécifiées dans les mappings ("ID" et "NAME").

La forme suivante n'est pas vulnérable à la duplication des noms de colonnes :

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*} FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
    .addEntity("cat", Cat.class)
    .addEntity("mother", Cat.class)
```

Cette requête spécifie :

- la requête SQL, avec des réceptacles pour que Hibernate injecte les alias de colonnes
- les entités retournées par la requête

Les notations {cat.*} et {mother.*} utilisées ci-dessus sont un équivalent à 'toutes les propriétés'. Alternativement, vous pouvez lister les colonnes explicitement, mais même dans ce cas, nous laissons Hibernate injecter les alias de colonne pour chaque propriété. Le paramètre fictif pour un alias de colonne est simplement le nom de la propriété qualifié par l'alias de la table. Dans l'exemple suivant, nous récupérons les Cats et leur mère depuis une table différente (cat_log) de celle déclarée dans les mappings. Notez que nous pouvons aussi utiliser les alias de propriété dans la clause where si désiré.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
    "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
    "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
    .addEntity("cat", Cat.class)
    .addEntity("mother", Cat.class).list()
```

17.1.4.1. Références d'alias et de propriété

Pour la plupart des cas précédents, l'injection d'alias est requise, mais pour les requêtes relatives à des mappings plus complexes, comme les propriétés composites, les discriminants d'héritage, les collections etc., il y a des alias spécifiques à utiliser pour permettre à Hibernate l'injection des alias appropriés.

Le tableau suivant montre les diverses possibilités d'utilisation d'injection d'alias. Note : les noms d'alias dans le résultat sont des exemples, chaque alias aura un nom unique et probablement différent lorsqu'ils seront utilisés.

Tableau 17.1. Nom d'injection d'alias

Description	Syntaxe	Exemple
Une propriété simple	{[aliasname]. [propertyname]}	A_NAME as {item.name}

Description	Syntaxe	Exemple
Une propriété composite	{[aliasname].[componentname].[propertyname]}	CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}
Discriminateur d'une entité	{[aliasname].class}	DISC as {item.class}
Toutes les propriétés d'une entité	{[aliasname].*}	{item.*}
La clé d'une collection	{[aliasname].key}	ORGID as {coll.key}
L'id d'une collection	{[aliasname].id}	EMPID as {coll.id}
L'élément d'une collection	{[aliasname].element}	*ID as {coll.element}
Propriété de l'élément dans une collection	{[aliasname].element.[propertyname]}	NAME as {coll.element.name}
Toutes les propriétés d'un élément dans la collection	{[aliasname].element.*}	{coll.*}.element.*}
All properties of the collection	{[aliasname].*}	{coll.*}

17.1.5. Retour d'entités non gérées

Il est possible d'appliquer un `ResultTransformer` à une requête native SQL. Ce qui permet, par exemple, de retourner des entités non gérées.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
    .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

Cette requête spécifie :

- la chaîne de requêtes SQL
- un transformateur de résultat

La requête précédente retournera la liste de `CatDTO` qui ont été instanciés et dans lesquels les valeurs de `NAME` et `BIRTHNAME` auront été injectées dans leurs propriétés ou champs correspondants.

17.1.6. Gérer l'héritage

Les requêtes natives SQL qui interrogent des entités mappées en tant que part d'un héritage doivent inclure toutes les propriétés de la classe de base et de toutes ses sous classes.

17.1.7. Paramètres

Les requêtes natives SQL supportent aussi bien les paramètres de position que les paramètres nommés :

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

17.2. Requêtes SQL nommées

Les requêtes SQL nommées peuvent être définies dans le document de mapping et appelées exactement de la même manière qu'une requête HQL nommée. Dans ce cas, nous *n'avons pas besoin* d'appeler `addEntity()`.

```
<sql-query name="persons">
  <return alias="person" class="eg.Person"/>
  SELECT person.NAME AS {person.name},
         person.AGE AS {person.age},
         person.SEX AS {person.sex}
  FROM PERSON person
  WHERE person.NAME LIKE :namePattern
</sql-query>
>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

Les éléments `<return-join>` et `<load-collection>` sont respectivement utilisés pour lier des associations et définir des requêtes qui initialisent des collections,

```
<sql-query name="personsWith">
  <return alias="person" class="eg.Person"/>
  <return-join alias="address" property="person.mailingAddress"/>
  SELECT person.NAME AS {person.name},
         person.AGE AS {person.age},
         person.SEX AS {person.sex},
```



```

        address.STREET AS {address.street},
        address.CITY AS {address.city},
        address.STATE AS {address.state},
        address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
>

```

Une requête SQL nommée peut retourner une valeur scalaire. Vous devez spécifier l'alias de colonne et le type Hibernate utilisant l'élément `<return-scalar>` :

```

<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
>

```

Vous pouvez externaliser les informations de mapping des résultats dans un élément `<resultset>` pour soit les réutiliser dans différentes requêtes nommées, soit à travers l'API `setResultSetMapping()`.

```

<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
>

```

Vous pouvez également utiliser les informations de mapping de l'ensemble de résultats dans vos fichiers hbm directement dans le code java.

```
List cats = sess.createSQLQuery(
    "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
)
.setResultSetMapping("catAndKitten")
.list();
```

17.2.1. Utilisation de return-property pour spécifier explicitement les noms des colonnes/alias

Avec `<return-property>` vous pouvez explicitement dire à Hibernate quels alias de colonne utiliser, plutôt que d'employer la syntaxe `{ }` pour laisser Hibernate injecter ses propres alias. Par exemple :

```
<sql-query name="mySqlQuery">
  <return alias="person" class="eg.Person">
    <return-property name="name" column="myName" />
    <return-property name="age" column="myAge" />
    <return-property name="sex" column="mySex" />
  </return>
  SELECT person.NAME AS myName,
         person.AGE AS myAge,
         person.SEX AS mySex,
  FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` fonctionne aussi avec de multiples colonnes. Cela résout une limitation de la syntaxe `{ }` qui ne permet pas une fine granularité des propriétés multi-colonnes.

```
<sql-query name="organizationCurrentEmployments">
  <return alias="emp" class="Employment">
    <return-property name="salary">
      <return-column name="VALUE" />
      <return-column name="CURRENCY" />
    </return-property>
    <return-property name="endDate" column="myEndDate" />
  </return>
  SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
  STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
  REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
  FROM EMPLOYMENT
  WHERE EMPLOYER = :id AND ENDDATE IS NULL
  ORDER BY STARTDATE ASC
</sql-query>
>
```

Notez que dans cet exemple nous avons utilisé `<return-property>` en combinaison avec la syntaxe `{ }` pour l'injection. Cela autorise les utilisateurs à choisir comment ils veulent référencer les colonnes et les propriétés.

Si votre mapping a un discriminant vous devez utiliser `<return-discriminator>` pour spécifier la colonne discriminante.

17.2.2. Utilisation de procédures stockées pour les requêtes

Hibernate 3 introduit le support des requêtes via les procédures stockées et les fonctions. La documentation suivante est valable pour les deux. Les procédures stockées/fonctions doivent retourner un ensemble de résultats en tant que premier paramètre sortant (out-parameter") pour être capable de fonctionner avec Hibernate. Voici un exemple d'une telle procédure stockée en Oracle 9 et version supérieure :

```
CREATE OR REPLACE FUNCTION selectAllEmployments
RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
    SELECT EMPLOYEE, EMPLOYER,
    STARTDATE, ENDDATE,
    REGIONCODE, EID, VALUE, CURRENCY
    FROM EMPLOYMENT;
    RETURN st_cursor;
END;
```

Pour utiliser cette requête dans Hibernate vous avez besoin de la mapper via une requête nommée.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
>
```

Notez que les procédures stockées ne retournent, pour le moment, que des scalaires et des entités. `<return-join>` et `<load-collection>` ne sont pas supportés.

17.2.2.1. Règles/limitations lors de l'utilisation des procédures stockées

Pour utiliser des procédures stockées avec Hibernate, les procédures doivent suivre certaines règles. Si elles ne suivent pas ces règles, elles ne sont pas utilisables avec Hibernate. Si néanmoins, vous désirez utiliser ces procédures vous devez les exécuter via `session.connection()`. Les règles sont différentes pour chaque base de données, puisque les vendeurs de base de données ont des sémantiques/syntaxes différentes pour les procédures stockées.

Les requêtes de procédures stockées ne peuvent pas être paginées avec `setFirstResult()/setMaxResults()`.

La forme d'appel recommandée est le SQL92 standard : `{ ? = call functionName(<parameters>) }` or `{ ? = call procedureName(<parameters>) }`. La syntaxe d'appel native n'est pas supportée.

Pour Oracle les règles suivantes sont applicables :

- La procédure doit retourner un ensemble de résultats. Le premier paramètre d'une procédure doit être un `OUT` qui retourne un ensemble de résultats. Ceci est effectué en retournant un `SYS_REFCURSOR` dans Oracle 9 ou 10. Dans Oracle vous avez besoin de définir un type `REFCURSOR`, consultez la documentation Oracle.

Pour Sybase ou MS SQL server les règles suivantes sont applicables :

- La procédure doit retourner un ensemble de résultats. Notez que comme ces serveurs peuvent retourner de multiples ensembles de résultats et mettre à jour des compteurs, Hibernate itérera les résultats et prendra le premier résultat qui est un ensemble de résultats comme valeur de retour. Tout le reste sera ignoré.
- Si vous pouvez activer `SET NOCOUNT ON` dans votre procédure, elle sera probablement plus efficace, mais ce n'est pas une obligation.

17.3. SQL personnalisé pour créer, mettre à jour et effacer

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or individual column level. This section describes statement overrides. For columns, see [Section 5.7, « Column read and write expressions »](#).

The class and collection persisters in Hibernate already contain a set of configuration time generated strings (insertsql, deletesql, updatesql etc.). The mapping tags `<sql-insert>`, `<sql-delete>`, and `<sql-update>` override these strings:

```
<class name="Person">
  <id name="id">
```

```

        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert
>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update
>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete
>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
>

```

Le SQL est directement exécuté dans votre base de données, donc vous êtes libre d'utiliser le dialecte que vous souhaitez. Cela réduira bien sûr la portabilité de votre mappage si vous utilisez du SQL spécifique à votre base de données.

Les procédures stockées sont supportées si l'attribut `callable` est paramétré :

```

<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true"
>{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true"
>{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true"
>{? = call updatePerson (?, ?)}</sql-update>
</class>
>

```

L'ordre des paramètres de position est essentiel, car ils doivent être dans la séquence attendue par Hibernate.

Vous pouvez voir l'ordre attendu en activant la journalisation de débogage pour le niveau `org.hibernate.persister.entity`. Avec ce niveau activé, Hibernate imprimera le SQL statique qui est utilisé pour créer, mettre à jour, supprimer, etc. des entités. Pour voir la séquence attendue, rappelez-vous de ne pas inclure votre SQL personnalisé dans les fichiers de mappage car cela surchargera le SQL statique généré par Hibernate.

Les procédures stockées sont dans la plupart des cas (lire : il vaut mieux le faire) requises pour retourner le nombre de lignes insérées/mises à jour/supprimées, puisque Hibernate vérifie plusieurs fois le succès de l'expression. Hibernate inscrit toujours le premier paramètre de l'expression comme un paramètre de sortie numérique pour les opérations CUD :

```

CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
RETURN NUMBER IS
BEGIN

```

```
update PERSON
set
    NAME = uname,
where
    ID = uid;

return SQL%ROWCOUNT;

END updatePerson;
```

17.4. SQL personnalisé pour le chargement

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in [Section 5.7, « Column read and write expressions »](#) or at the statement level. Here is an example of a statement level override:

```
<sql-query name="person">
  <return alias="pers" class="Person" lock-mode="upgrade"/>
  SELECT NAME AS {pers.name}, ID AS {pers.id}
  FROM PERSON
  WHERE ID=?
  FOR UPDATE
</sql-query>
>
```

Ceci est juste une déclaration de requête nommée, comme vu précédemment. Vous pouvez référencer cette requête nommée dans un mappage de classe :

```
<class name="Person">
  <id name="id">
    <generator class="increment"/>
  </id>
  <property name="name" not-null="true"/>
  <loader query-ref="person"/>
</class>
>
```

Ceci fonctionne même avec des procédures stockées.

Vous pouvez même définir une requête pour le chargement d'une collection :

```
<set name="employments" inverse="true">
  <key/>
  <one-to-many class="Employment"/>
  <loader query-ref="employments"/>
</set>
>
```

```
<sql-query name="employments">
  <load-collection alias="emp" role="Person.employments"/>
  SELECT {emp.*}
  FROM EMPLOYMENT emp
  WHERE EMPLOYER = :id
  ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
>
```

Vous pourriez même définir un chargeur d'entité qui charge une collection par jointure :

```
<sql-query name="person">
  <return alias="pers" class="Person"/>
  <return-join alias="emp" property="pers.employments"/>
  SELECT NAME AS {pers.*}, {emp.*}
  FROM PERSON pers
  LEFT OUTER JOIN EMPLOYMENT emp
    ON pers.ID = emp.PERSON_ID
  WHERE ID=?
</sql-query>
>
```


Filtrer les données

Hibernate3 fournit une nouvelle approche innovatrice pour manipuler des données avec des règles de "visibilité". Un *filtre Hibernate* est un filtre global, nommé, paramétré qui peut être activé ou désactivé pour une session Hibernate particulière.

18.1. Filtres Hibernate

Hibernate3 ajoute la capacité de prédéfinir des critères de filtre et d'attacher ces filtres à une classe ou à une collection. Un critère de filtre est la faculté de définir une clause de restriction très similaire à l'attribut "where" existant disponible sur une classe et divers éléments d'une collection. Par ailleurs ces conditions de filtre peuvent être paramétrées. L'application peut alors prendre la décision à l'exécution si des filtres donnés doivent être activés et quels doivent être leurs paramètres. Des filtres peuvent être utilisés comme des vues de base de données, mais paramétrées dans l'application.

Afin d'utiliser des filtres, ils doivent d'abord être définis, puis attachés aux éléments de mapping appropriés. Pour définir un filtre, utilisez l'élément `<filter-def/>` dans un élément `<hibernate-mapping/>` :

```
<filter-def name="myFilter">
  <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

Puis, ce filtre peut être attaché à une classe :

```
<class name="myClass" ...>
  ...
  <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</class>
```

Ou bien, à une collection :

```
<set ...>
  <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</set>
```

Ou même encore, aux deux (ou à plusieurs de chaque) en même temps.

Les méthodes sur Session sont : `enableFilter(String filterName)`, `getEnabledFilter(String filterName)`, et `disableFilter(String filterName)`. Par

défaut, les filtres *ne sont pas* activés pour une session donnée ; ils doivent être explicitement activés en appelant la méthode `Session.enableFilter()`, laquelle retourne une instance de l'interface `Filter`. Utiliser le simple filtre défini ci-dessus ressemblerait à :

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

Notez que des méthodes sur l'interface `org.hibernate.Filter` autorisent le chaînage de beaucoup de méthodes communes à Hibernate.

Un exemple complet, utilisant des données temporelles avec une structure de date d'enregistrement effectif :

```
<filter-def name="effectiveDate">
  <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
  ...
  <many-to-one name="department" column="dept_id" class="Department"/>
  <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
  <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
  ...
  <!--
    Note that this assumes non-terminal records have an eff_end_dt set to
    a max db date for simplicity-sake
  -->
  <filter name="effectiveDate"
    condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
  ...
  <set name="employees" lazy="true">
    <key column="dept_id"/>
    <one-to-many class="Employee"/>
    <filter name="effectiveDate"
      condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
  </set>
</class>
>
```

Puis, afin de s'assurer que vous pouvez toujours récupérer les enregistrements actuellement effectifs, activez simplement le filtre sur la session avant de récupérer des données des employés :

```
Session session = ...;
session.enableFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary
> :targetSalary")
    .setLong("targetSalary", new Long(1000000))
```

```
.list();
```

Dans le HQL ci-dessus, bien que nous ayons seulement mentionné une contrainte de salaire sur les résultats, à cause du filtre activé, la requête retournera seulement les employés actuellement actifs qui ont un salaire supérieur à un million de dollars.

A noter : si vous prévoyez d'utiliser des filtres avec des jointures externes (soit à travers HQL, soit par le chargement), faites attention à la direction de l'expression de condition. Il est plus sûr de la positionner pour les jointures externes à gauche ; en général, placez le paramètre d'abord, suivi du(des) nom(s) de colonne après l'opérateur.

Après avoir été défini, un filtre peut être attaché à de nombreuses entités et/ou des collections, chacune avec sa propre condition. Cela peut être fastidieux quand les conditions sont les mêmes à chaque fois. Ainsi `<filter-def/>` permet de définir une condition par défaut, soit en tant qu'attribut, soit comme CDATA.

```
<filter-def name="myFilter" condition="abc
> xyz "
>...</filter-def>
<filter-def name="myOtherFilter"
>abc=xyz</filter-def
>
```

Cette condition par défaut sera alors utilisée à chaque fois que le filtre est attaché à quelque chose sans spécifier la condition. Notez que cela signifie que vous pouvez fournir une condition spécifique en tant que faisant partie de la pièce attachée du filtre qui surcharge la condition par défaut dans ce cas particulier.

Mappage XML

XML Mapping is an experimental feature in Hibernate 3.0 and is currently under active development.

19.1. Travailler avec des données XML

Hibernate vous laisse travailler avec des données XML persistantes de la même manière que vous travaillez avec des POJO persistants. Un arbre XML peut être vu comme une autre manière de représenter les données relationnelles au niveau objet, à la place des POJO.

Hibernate supporte dom4j en tant qu'API pour la manipulation des arbres XML. Vous pouvez écrire des requêtes qui récupèrent des arbres dom4j à partir de la base de données, et avoir toutes les modifications que vous faites sur l'arbre automatiquement synchronisées dans la base de données. Vous pouvez même prendre un document XML, l'analyser en utilisant dom4j, et l'écrire dans la base de données via les opérations basiques de Hibernate : `persist()`, `saveOrUpdate()`, `merge()`, `delete()`, `replicate()` (merge n'est pas encore supporté).

Cette fonctionnalité a plusieurs applications dont l'import/export de données, l'externalisation de données d'entités via JMS ou SOAP et les rapports XSLT.

Un simple mappage peut être utilisé pour simultanément mapper les propriétés d'une classe et les noeuds d'un document XML vers la base de données, ou, s'il n'y a pas de classe à mapper, il peut être utilisé juste pour mapper le XML.

19.1.1. Spécifier le mappage XML et le mappage d'une classe ensemble

Voici un exemple de mappage d'un POJO et du XML simultanément :

```
<class name="Account"
      table="ACCOUNTS"
      node="account">

  <id name="accountId"
      column="ACCOUNT_ID"
      node="@id" />

  <many-to-one name="customer"
      column="CUSTOMER_ID"
      node="customer/@id"
      embed-xml="false" />

  <property name="balance"
      column="BALANCE"
      node="balance" />

  ...
```

```
</class  
>
```

19.1.2. Spécifier seulement un mappage XML

Voici un exemple dans lequel il n'y a pas de classe POJO :

```
<class entity-name="Account"  
      table="ACCOUNTS"  
      node="account">  
  
  <id name="id"  
      column="ACCOUNT_ID"  
      node="@id"  
      type="string"/>  
  
  <many-to-one name="customerId"  
      column="CUSTOMER_ID"  
      node="customer/@id"  
      embed-xml="false"  
      entity-name="Customer"/>  
  
  <property name="balance"  
      column="BALANCE"  
      node="balance"  
      type="big_decimal"/>  
  
  ...  
  
</class  
>
```

Ce mappage vous permet d'accéder aux données comme un arbre dom4j, ou comme un graphe de paire nom/valeur de propriété (Maps java). Les noms des propriétés sont des constructions purement logiques qui peuvent être référées dans des requêtes HQL.

19.2. Métadonnées du mappage XML

Plusieurs éléments du mappage Hibernate acceptent l'attribut `node`. Ceci vous permet de spécifier le nom d'un attribut XML ou d'un élément qui contient la propriété ou les données de l'entité. Le format de l'attribut `node` doit être un des suivants :

- "element-name" - mappe vers l'élément XML nommé
- "@attribute-name" - mappe vers l'attribut XML nommé
- "." - mappe vers le parent de l'élément
- "element-name/@attribute-name" - mappe vers l'élément nommé de l'attribut nommé

Pour des collections et de simples associations valuées, il y a un attribut `embed-xml` supplémentaire. Si `embed-xml="true"`, qui est la valeur par défaut, l'arbre XML pour l'entité associée (ou la collection des types de valeurs) sera embarquée directement dans l'arbre XML

pour l'entité qui possède l'association. Sinon, si `embed-xml="false"`, alors seule la valeur de l'identifiant référencé apparaîtra dans le XML pour de simples associations de points, et les collections n'apparaîtront pas.

Faire attention à ne pas laisser `embed-xml="true"` pour trop d'associations, puisque XML ne traite pas bien les liens circulaires.

```
<class name="Customer"
      table="CUSTOMER"
      node="customer">

  <id name="id"
      column="CUST_ID"
      node="@id"/>

  <map name="accounts"
      node="."
      embed-xml="true">
    <key column="CUSTOMER_ID"
        not-null="true"/>
    <map-key column="SHORT_DESC"
        node="@short-desc"
        type="string"/>
    <one-to-many entity-name="Account"
        embed-xml="false"
        node="account"/>
  </map>

  <component name="name"
      node="name">
    <property name="firstName"
        node="first-name"/>
    <property name="initial"
        node="initial"/>
    <property name="lastName"
        node="last-name"/>
  </component>

  ...

</class>
>
```

Dans ce cas, nous avons décidé d'embarquer la collection d'identifiants de compte, mais pas les données actuelles du compte. La requête HQL suivante :

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

devrait retourner l'ensemble de données suivant :

```
<customer id="123456789">
```

```
<account short-desc="Savings"
>987632567</account>
  <account short-desc="Credit Card"
>985612323</account>
    <name>
      <first-name
>Gavin</first-name>
      <initial
>A</initial>
      <last-name
>King</last-name>
    </name>
    ...
</customer>
>
```

Si vous positionnez `embed-xml="true"` sur le mappage `<one-to-many>`, les données ressembleraient à ce qui suit :

```
<customer id="123456789">
  <account id="987632567" short-desc="Savings">
    <customer id="123456789"/>
    <balance
>100.29</balance>
  </account>
  <account id="985612323" short-desc="Credit Card">
    <customer id="123456789"/>
    <balance
>-2370.34</balance>
  </account>
  <name>
    <first-name
>Gavin</first-name>
    <initial
>A</initial>
    <last-name
>King</last-name>
  </name>
  ...
</customer>
>
```

19.3. Manipuler des données XML

Relisons et mettons à jour des documents XML dans l'application. Nous effectuons cela en obtenant une session dom4j :

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();
```



```
List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

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

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

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

Il est extrêmement utile de combiner cette fonctionnalité avec l'opération `replicate()` de Hibernate pour implémenter des imports/exports de données XML.

Améliorer les performances

20.1. Stratégies de chargement

Une *stratégie de chargement* est une stratégie que Hibernate va utiliser pour récupérer des objets associés si l'application a besoin de naviguer à travers une association. Les stratégies de chargement peuvent être déclarées dans les méta-données de l'outil de mappage objet relationnel, ou surchargées par une requête de type HQL ou `Criteria` particulière.

Hibernate3 définit les stratégies de chargement suivantes :

- *Chargement par jointure* - Hibernate récupère l'instance associée ou la collection dans un même `SELECT`, en utilisant un `OUTER JOIN`.
- *Chargement par select* - Un second `SELECT` est utilisé pour récupérer l'instance associée à l'entité ou à la collection. À moins que vous ne désactiviez explicitement le chargement différé en spécifiant `lazy="false"`, ce second select ne sera exécuté que lorsque vous accéderez réellement à l'association.
- *Chargement par sous-select* - Un second `SELECT` est utilisé pour récupérer les associations pour toutes les entités récupérées dans une requête ou un chargement préalable. A moins que vous ne désactiviez explicitement le chargement différé en spécifiant `lazy="false"`, ce second select ne sera exécuté que lorsque vous accéderez réellement à l'association.
- *Chargement par lot* - Il s'agit d'une stratégie d'optimisation pour le chargement par select - Hibernate récupère un lot d'instances ou de collections en un seul `SELECT` en spécifiant une liste de clés primaires ou de clés étrangères.

Hibernate fait également la distinction entre :

- *Chargement immédiat* - Une association, une collection ou un attribut est chargé immédiatement lorsque l'objet auquel appartient cet élément est chargé.
- *Chargement différé d'une collection* - Une collection est chargée lorsque l'application invoque une méthode sur cette collection (il s'agit du mode de chargement par défaut pour les collections).
- *Chargement "super différé" d'une collection* - On accède aux éléments de la collection depuis la base de données lorsque c'est nécessaire. Hibernate essaie de ne pas charger toute la collection en mémoire sauf si cela est absolument nécessaire (bien adapté aux très grandes collections).
- *Chargement par proxy* - Une association vers un seul objet est chargée lorsqu'une méthode autre que le getter sur l'identifiant est appelée sur l'objet associé.
- *Chargement "sans proxy"* - une association vers un seul objet est chargée lorsque l'on accède à la variable d'instance. Par rapport au chargement par proxy, cette approche est moins

différée (l'association est quand même chargée même si on n'accède qu'à l'identifiant) mais plus transparente car il n'y a pas de proxy visible dans l'application. Cette approche requiert une instrumentation du bytecode à la compilation et est rarement nécessaire.

- *Chargement différé des attributs* - Un attribut ou un objet associé seul est chargé lorsque l'on accède à la variable d'instance. Cette approche requiert une instrumentation du bytecode à la compilation et est rarement nécessaire.

Nous avons ici deux notions orthogonales : *quand* l'association est chargée et *comment* (quelle requête SQL est utilisée). Il ne faut pas les confondre. Le mode de chargement `fetch` est utilisé pour améliorer les performances. On peut utiliser le mode `lazy` pour définir un contrat sur quelles données sont toujours accessibles sur toute instance détachée d'une classe particulière.

20.1.1. Travailler avec des associations chargées en différé

Par défaut, Hibernate3 utilise le chargement différé par select pour les collections et le chargement différé par proxy pour les associations vers un seul objet. Ces valeurs par défaut sont valables pour la plupart des associations dans la plupart des applications.

Si vous définissez `hibernate.default_batch_fetch_size`, Hibernate va utiliser l'optimisation du chargement par lot pour le chargement différé (cette optimisation peut aussi être activée à un niveau de granularité plus fin).

L'accès à une association définie comme "différé", hors du contexte d'une session Hibernate ouverte, entraîne une exception. Par exemple :

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts"); // Error!
```

Étant donné que la collection des permissions n'a pas été initialisée avant de fermer la `Session`, la collection n'est pas capable de charger son état. *Hibernate ne supporte pas le chargement différé pour des objets détachés*. La solution à ce problème est de déplacer le code qui lit à partir de la collection avant le "commit" de la transaction.

Une autre alternative est d'utiliser une collection ou une association non "différée" en spécifiant `lazy="false"` dans le mappage de l'association. Cependant il est prévu que le chargement différé soit utilisé pour quasiment toutes les collections ou associations. Si vous définissez trop d'associations non "différées" dans votre modèle objet, Hibernate va finir par devoir charger toute la base de données en mémoire à chaque transaction.

Par ailleurs, on veut souvent choisir un chargement par jointure (qui est par défaut non différé) à la place du chargement par select dans une transaction particulière. Nous allons maintenant voir comment adapter les stratégies de chargement. Dans Hibernate3 les mécanismes pour choisir une stratégie de chargement sont identiques que l'on ait une association vers un objet simple ou vers une collection.

20.1.2. Personnalisation des stratégies de chargement

Le chargement par select (mode par défaut) est très vulnérable au problème du N+1 selects, ainsi vous souhaitez peut-être activer le chargement par jointure dans les fichiers de mappage :

```
<set name="permissions"
    fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set>
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

La stratégie de chargement définie à l'aide du mot `fetch` dans les fichiers de mappage affecte :

- La récupération via `get()` ou `load()`
- La récupération implicite lorsque l'on navigue à travers une association
- Les requêtes par `Criteria`
- Les requêtes HQL si l'on utilise le chargement par `subselect`

Quelle que soit la stratégie de chargement que vous utilisez, la partie du graphe d'objets, non-différée, sera chargée en mémoire. Cela peut mener à l'exécution de plusieurs selects successifs pour une seule requête HQL.

On n'utilise pas souvent les documents de mappage pour adapter le chargement. En revanche, on conserve le comportement par défaut et on le surcharge pour une transaction particulière en utilisant `left join fetch` dans les requêtes HQL. Cela indique à Hibernate de charger l'association de manière agressive lors du premier select en utilisant une jointure externe. Dans la requête API `Criteria` vous utiliserez la méthode `setFetchMode(FetchMode.JOIN)`.

S'il vous arrive de vouloir changer la stratégie de chargement utilisée par utilisée par `get()` ou `load()`, vous pouvez juste utiliser une requête de type `Criteria` comme par exemple :

```
User user = (User) session.createCriteria(User.class)
    .setFetchMode("permissions", FetchMode.JOIN)
    .add( Restrictions.idEq(userId) )
```

```
.uniqueResult();
```

Il s'agit de l'équivalent pour Hibernate de ce que d'autres outils de mappage appellent un "fetch plan" ou "plan de chargement".

Une autre manière complètement différente d'éviter le problème des N+1 selects est d'utiliser le cache de second niveau.

20.1.3. Proxies pour des associations vers un seul objet

Le chargement différé des collections est implémenté par Hibernate qui utilise ses propres implémentations pour des collections persistantes. Si l'on veut un chargement différé pour des associations vers un seul objet, il faut utiliser un autre mécanisme. L'entité qui est pointée par l'association doit être masquée derrière un proxy. Hibernate implémente l'initialisation différée des proxies sur des objets persistants via une mise à jour à chaud du bytecode (à l'aide de l'excellente librairie CGLIB).

Par défaut, Hibernate génère des proxies (au démarrage) pour toutes les classes persistantes et les utilise pour activer le chargement différé des associations `many-to-one` et `one-to-one`.

Le fichier de mappage peut déclarer une interface à utiliser comme interface de proxy pour cette classe à l'aide de l'attribut `proxy`. Par défaut Hibernate utilise une sous-classe de la classe persistante. *Il faut que les classes pour lesquelles on ajoute un proxy implémentent un constructeur par défaut avec au minimum une visibilité de paquetage. Ce constructeur est recommandé pour toutes les classes persistantes !.*

Il y a quelques précautions à prendre lorsque l'on étend cette approche à des classes polymorphiques, par exemple :

```
<class name="Cat" proxy="Cat">
    .....
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
>
```

Tout d'abord, les instances de `Cat` ne pourront jamais être "castées" en `DomesticCat`, même si l'instance sous-jacente est une instance de `DomesticCat` :

```
Cat cat = (Cat) session.load(Cat.class, id); // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;      // Error!
    ....
}
```

Deuxièmement, il est possible de casser la notion de `==` des proxies.

```
Cat cat = (Cat) session.load(Cat.class, id);           // instantiate a Cat proxy
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id); // acquire new DomesticCat proxy!
System.out.println(cat==dc);                          // false
```

Cette situation n'est pas si mauvaise qu'il n'y paraît. Même si nous avons deux références à deux objets proxies différents, l'instance sous-jacente sera quand même le même objet :

```
cat.setWeight(11.0); // hit the db to initialize the proxy
System.out.println( dc.getWeight() ); // 11.0
```

Troisièmement, vous ne pourrez pas utiliser un proxy CGLIB pour une classe `final` ou pour une classe contenant la moindre méthode `final`.

Enfin, si votre objet persistant obtient une quelconque ressource à l'instanciation (par exemple dans les initialiseurs ou dans le constructeur par défaut), alors ces ressources seront aussi obtenues par le proxy. La classe proxy est en réalité une sous-classe de la classe persistante.

Ces problèmes sont tous dus aux limitations fondamentales du modèle d'héritage unique de Java. Si vous souhaitez éviter ces problèmes, vos classes persistantes doivent chacune implémenter une interface qui déclare ses méthodes métier. Vous devriez alors spécifier ces interfaces dans le fichier de mappage : `CatImpl` implémente l'interface `Cat` et `DomesticCatImpl` implémente l'interface `DomesticCat`. Par exemple :

```
<class name="CatImpl" proxy="Cat">
    .....
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
>
```

Tout d'abord, les instances de `Cat` et de `DomesticCat` peuvent être retournées par `load()` ou par `iterate()`.

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();
Cat fritz = (Cat) iter.next();
```



Remarque

`list()` ne retourne pas les proxies normalement.

Les relations sont aussi initialisées en différé. Ceci signifie que vous devez déclarer chaque propriété comme étant de type `Cat`, et non `CatImpl`.

Certaines opérations ne nécessitent *pas* l'initialisation du proxy :

- `equals()`, si la classe persistante ne surcharge pas `equals()`
- `hashCode()`, si la classe persistante ne surcharge pas `hashCode()`
- La méthode getter de l'identifiant

Hibernate détectera les classes qui surchargent `equals()` ou `hashCode()`.

Eh choisissant `lazy="no-proxy"` au lieu de `lazy="proxy"` qui est la valeur par défaut, il est possible d'éviter les problèmes liés au transtypage. Il faudra alors une instrumentation du bytecode à la compilation et toutes les opérations résulteront immédiatement en une initialisation du proxy.

20.1.4. Initialisation des collections et des proxies

Une exception de type `LazyInitializationException` sera renvoyée par Hibernate si une collection ou un proxy non initialisé est accédé en dehors de la portée de la `Session`, par ex. lorsque l'entité à laquelle appartient la collection ou qui a une référence vers le proxy, est dans l'état "détaché".

Parfois, nous devons nous assurer qu'un proxy ou une collection est initialisé avant de fermer la `Session`. Bien sûr, nous pouvons toujours forcer l'initialisation en appelant par exemple `cat.getSex()` ou `cat.getKittens().size()`. Mais ceci n'est pas très lisible pour les personnes parcourant le code et n'est pas approprié pour le code générique.

Les méthodes statiques `Hibernate.initialize()` et `Hibernate.isInitialized()` fournissent à l'application un moyen de travailler avec des proxies ou des collections initialisés. `Hibernate.initialize(cat)` forcera l'initialisation d'un proxy de `cat`, si tant est que sa `Session` est ouverte. `Hibernate.initialize(cat.getKittens())` a le même effet sur la collection `kittens`.

Une autre option est de conserver la `Session` ouverte jusqu'à ce que toutes les collections et tous les proxies nécessaires aient été chargés. Dans certaines architectures applicatives, particulièrement celles où le code d'accès aux données via Hibernate et le code qui utilise ces données sont dans des couches applicatives différentes ou des processus physiques différents, il sera alors difficile de garantir que la `Session` est ouverte lorsqu'une collection est initialisée. Il y a deux moyens de maîtriser ce problème :

- Dans une application web, un filtre de servlet peut être utilisé pour fermer la `Session` uniquement lorsque la requête a été entièrement traitée, lorsque le rendu de la vue est fini (il s'agit du modèle *Vue de la session ouverte*). Bien sûr, cela demande plus d'attention à la bonne gestion des exceptions de l'application. Il est d'une importance vitale que la `Session`

soit fermée et la transaction terminée avant que l'on rende la main à l'utilisateur même si une exception survient durant le traitement de la vue. Voir le wiki Hibernate pour des exemples sur le modèle "Open Session in View".

- Dans une application avec une couche métier multiniveaux séparée, la couche contenant la logique métier doit "préparer" toutes les collections qui seront nécessaires à la couche web multiniveaux avant de retourner les données. Cela signifie que la couche métier doit charger toutes les données et retourner toutes les données déjà initialisées à la couche de présentation/web pour un cas d'utilisation donné. En général l'application appelle la méthode `Hibernate.initialize()` pour chaque collection nécessaire dans la couche web (cet appel doit être fait avant la fermeture de la session) ou bien récupère les collections de manière agressive à l'aide d'une requête HQL avec une clause `FETCH` ou à l'aide du mode `FetchMode.JOIN` pour une requête de type `Criteria`. Cela est en général plus facile si vous utilisez le modèle *Command* plutôt que *Session Facade*.
- Vous pouvez également attacher à une `Session` un objet chargé au préalable à l'aide des méthodes `merge()` ou `lock()` avant d'accéder aux collections (ou aux proxies) non initialisés. Non, Hibernate ne fait pas, et ne doit pas faire cela automatiquement car cela pourrait introduire une sémantique transactionnelle ad hoc.

Parfois, vous ne voulez pas initialiser une grande collection mais vous avez quand même besoin d'informations sur elle (comme sa taille) ou un sous-ensemble de ses données.

Vous pouvez utiliser un filtre de collection pour récupérer sa taille sans l'initialiser :

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

La méthode `createFilter()` est également utilisée pour récupérer efficacement des sous-ensembles d'une collection sans avoir besoin de l'initialiser dans son ensemble :

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

20.1.5. Utiliser le chargement par lot

Pour améliorer les performances, Hibernate peut utiliser le chargement par lot ce qui veut dire que Hibernate peut charger plusieurs proxies (ou collections) non initialisés en une seule requête lorsque l'on accède à l'un de ces proxies. Le chargement par lot est une optimisation intimement liée à la stratégie de chargement en différé par select. Il y a deux moyens d'activer le chargement par lot : au niveau de la classe et au niveau de la collection.

Le chargement par lot pour les classes/entités est plus simple à comprendre. Imaginez que vous ayez la situation suivante à l'exécution : vous avez 25 instances de `Cat` chargées dans une `Session`, chaque `Cat` a une référence à son `owner`, une `Person`. La classe `Person` est mappée avec un proxy, `lazy="true"`. Si vous itérez sur tous les cats et appelez `getOwner()` sur chacun

d'eux, Hibernate exécutera par défaut 25 `SELECT`, pour charger les owners (initialiser le proxy). Vous pouvez paramétrer ce comportement en spécifiant une `batch-size` (taille du lot) dans le mappage de `Person` :

```
<class name="Person" batch-size="10"
>...</class
>
```

Hibernate exécutera désormais trois requêtes, en chargeant respectivement 10, 10, et 5 entités.

Vous pouvez aussi activer le chargement par lot pour les collections. Par exemple, si chaque `Person` a une collection chargée en différé des `Cats`, et que 10 personnes sont actuellement chargées dans la `Session`, itérer sur toutes les persons générera 10 `SELECT` s, un pour chaque appel de `getCats()`. Si vous activez le chargement par lot pour la collection `cats` dans le mappage de `Person`, Hibernate pourra précharger les collections :

```
<class name="Person">
  <set name="cats" batch-size="3">
    ...
  </set>
</class
>
```

Avec une taille de lot `batch-size` de 8, Hibernate chargera respectivement des collections 3, 3, et 1 en quatre `SELECT` s. Encore une fois, la valeur de l'attribut dépend du nombre de collections non initialisées dans une `Session` particulière.

Le chargement par lot de collections est particulièrement utile si vous avez une arborescence imbriquée d'éléments, c'est-à-dire le le schéma facture de matériels typique. (Bien qu'un *sous ensemble* ou un *chemin matérialisé* soit probablement une meilleure option pour des arbres principalement en lecture.)

20.1.6. Utilisation du chargement par sous select

Si une collection en différé ou un proxy vers un objet doit être chargée, Hibernate va tous les charger en ré-exécutant la requête originale dans un sous select. Cela fonctionne de la même manière que le chargement par lot sans la possibilité de fragmenter le chargement.

20.1.7. Fetch profiles

Another way to affect the fetching strategy for loading associated objects is through something called a fetch profile, which is a named configuration associated with the `org.hibernate.SessionFactory` but enabled, by name, on the `org.hibernate.Session`. Once enabled on a `org.hibernate.Session`, the fetch profile will be in affect for that `org.hibernate.Session` until it is explicitly disabled.

So what does that mean? Well lets explain that by way of an example. Say we have the following mappings:

```
<hibernate-mapping>
  <class name="Customer">
    ...
    <set name="orders" inverse="true">
      <key column="cust_id"/>
      <one-to-many class="Order"/>
    </set>
  </class>
  <class name="Order">
    ...
  </class>
</hibernate-mapping>
>
```

Now normally when you get a reference to a particular customer, that customer's set of orders will be lazy meaning we will not yet have loaded those orders from the database. Normally this is a good thing. Now lets say that you have a certain use case where it is more efficient to load the customer and their orders together. One way certainly is to use "dynamic fetching" strategies via an HQL or criteria queries. But another option is to use a fetch profile to achieve that. Just add the following to your mapping:

```
<hibernate-mapping>
  ...
  <fetch-profile name="customer-with-orders">
    <fetch entity="Customer" association="orders" style="join"/>
  </fetch-profile>
</hibernate-mapping>
>
```

or even:

```
<hibernate-mapping>
  <class name="Customer">
    ...
    <fetch-profile name="customer-with-orders">
      <fetch association="orders" style="join"/>
    </fetch-profile>
  </class>
  ...
</hibernate-mapping>
>
```

Now the following code will actually load both the customer *and their orders*:

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" ); // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

Currently only join style fetch profiles are supported, but they plan is to support additional styles. See [HHH-3414](http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414) [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414] for details.

20.1.8. Utiliser le chargement en différé des propriétés

Hibernate3 supporte le chargement en différé de propriétés individuelles. La technique d'optimisation est également connue sous le nom de *fetch groups* (groupes de chargement). Il faut noter qu'il s'agit principalement d'une fonctionnalité marketing car en pratique l'optimisation de la lecture d'un enregistrement est beaucoup plus importante que l'optimisation de la lecture d'une colonne. Cependant, la restriction du chargement à certaines colonnes peut être pratique dans des cas extrêmes, lorsque des tables "legacy" possèdent des centaines de colonnes et que le modèle de données ne peut pas être amélioré.

Pour activer le chargement en différé d'une propriété, il faut mettre l'attribut `lazy` sur une propriété particulière du mappage :

```
<class name="Document">
  <id name="id">
    <generator class="native"/>
  </id>
  <property name="name" not-null="true" length="50"/>
  <property name="summary" not-null="true" length="200" lazy="true"/>
  <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
>
```

Le chargement en différé des propriétés requiert une instrumentation du bytecode lors de la compilation ! Si les classes persistantes ne sont pas instrumentées, Hibernate ignorera de manière silencieuse le mode en différé et retombera dans le mode de chargement immédiat.

Pour l'instrumentation du bytecode vous pouvez utiliser la tâche Ant suivante :

```
<target name="instrument" depends="compile">
  <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
    <classpath path="${jar.path}"/>
    <classpath path="${classes.dir}"/>
    <classpath refid="lib.class.path"/>
  </taskdef>

  <instrument verbose="true">
    <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
      <include name="*.class"/>
    </fileset>
  </instrument>
</target>
```

```

    </fileset>
  </instrument>
</target>
>

```

Une autre façon (meilleure ?) pour éviter de lire plus de colonnes que nécessaire au moins pour des transactions en lecture seule est d'utiliser les fonctionnalités de projection des requêtes HQL ou Criteria. Cela évite de devoir instrumenter le bytecode à la compilation et est certainement une solution préférable.

Vous pouvez forcer le mode de chargement agressif des propriétés en utilisant `fetch all properties` dans les requêtes HQL.

20.2. Le cache de second niveau

Une `Session` Hibernate est un cache de niveau transactionnel de données persistantes. Il est possible de configurer un cache de cluster ou de JVM (de niveau `SessionFactory`) défini classe par classe et collection par collection. Vous pouvez même utiliser votre choix de cache en implémentant le fournisseur associé. Faites attention, les caches ne sont jamais avertis des modifications faites dans la base de données par d'autres applications (ils peuvent cependant être configurés pour régulièrement expirer les données en cache).

Vous pouvez choisir une autre implémentation en spécifiant le nom de la classe qui implémente `org.hibernate.cache.CacheProvider` en utilisant la propriété `hibernate.cache.provider_class`. Hibernate est accompagné de plusieurs intégrations imbriquées avec des fournisseurs de cache open-source (listés ci-dessous) ; par ailleurs vous pouvez implémenter votre propre fournisseur et le brancher comme indiqué ci-dessus. Notez que les versions antérieures à 3.2, utilisaient par défaut le EhCache comme le fournisseur de cache par défaut, ce qui n'est plus le cas.

Tableau 20.1. Fournisseurs de cache

Cache	Classe pourvoyeuse	Type	Cluster sécurisé	Cache de requêtes supporté
Table de hachage (ne pas utiliser en production)	<code>org.hibernate.cache.HashtableCacheProvider</code>	mémoire	yes	
EHCache	<code>org.hibernate.cache.EhCacheProvider</code>	mémoire, disque	yes	
OSCache	<code>org.hibernate.cache.OSCacheProvider</code>	mémoire, disque	yes	

Cache	Classe pourvoyeuse	Type	Cluster sécurisé	Cache de requêtes supporté
SwarmCache	org.hibernate.cache.SwarmCacheProvider	en cluster (multicast ip)	oui (invalidation de cluster)	
JBoss Cache 1.x	org.hibernate.cache.TreeCacheProvider	en cluster (multicast ip), transactionnel	oui (réplication)	oui (horloge sync. nécessaire)
JBoss Cache 2	org.hibernate.cache.jbc.JBossCacheProvider2Full	en cluster (multicast ip), transactionnel	oui (replication ou invalidation)	oui (horloge sync. nécessaire)

20.2.1. Mappages de Cache

L'élément `<cache>` d'une classe ou d'une collection a la forme suivante :

```
<cache
  usage="transactional|read-write|nonstrict-read-write|read-only"
  region="RegionName"
  include="all|non-lazy"
/>
```

- ❶ `usage` (requis) spécifie la stratégie de cache : transactionnel, lecture-écriture, lecture-écriture non stricte OU lecture seule
- ❷ `region` (optionnel, par défaut il s'agit du nom de la classe ou du nom de rôle de la collection) : spécifie le nom de la région du cache de second niveau
- ❸ `include` (optionnel, par défaut `all`) `non-lazy` : spécifie que les propriétés des entités mappées avec `lazy="true"` ne doivent pas être mises en cache lorsque le chargement en différé des attributs est activé.

Alternativement (voir préférentiellement), vous pouvez spécifier les éléments `<class-cache>` et `<collection-cache>` dans `hibernate.cfg.xml`.

L'attribut `usage` spécifie une *stratégie de concurrence d'accès au cache*.

20.2.2. Stratégie : lecture seule

Si votre application a besoin de lire mais ne modifie jamais les instances d'une classe, un cache `read-only` peut être utilisé. C'est la stratégie la plus simple et la plus performante. Elle est même parfaitement sûre dans un cluster.

```
<class name="eg.Immutable" mutable="false">
  <cache usage="read-only"/>
  ....
</class>
>
```

20.2.3. Stratégie : lecture/écriture

Si l'application a besoin de mettre à jour des données, un cache `read-write` peut être approprié. Cette stratégie ne devrait jamais être utilisée si votre application nécessite un niveau d'isolation transactionnelle sérialisable. Si le cache est utilisé dans un environnement JTA, vous devez spécifier la propriété `hibernate.transaction.manager_lookup_class`, fournissant une stratégie pour obtenir le JTA `TransactionManager`. Dans d'autres environnements, vous devriez vous assurer que la transaction est terminée à l'appel de `Session.close()` ou `Session.disconnect()`. Si vous souhaitez utiliser cette stratégie dans un cluster, vous devriez vous assurer que l'implémentation de cache utilisée supporte le verrouillage, ce que ne font *pas* les pourvoyeurs caches fournis.

```
<class name="eg.Cat" .... >
  <cache usage="read-write"/>
  ....
  <set name="kittens" ... >
    <cache usage="read-write"/>
    ....
  </set>
</class>
>
```

20.2.4. Stratégie : lecture/écriture non stricte

Si l'application a besoin de mettre à jour les données de manière occasionnelle (il est très peu probable que deux transactions essaient de mettre à jour le même élément simultanément) et si une isolation transactionnelle stricte n'est pas nécessaire, un cache `nonstrict-read-write` peut être approprié. Si le cache est utilisé dans un environnement JTA, vous devez spécifier `hibernate.transaction.manager_lookup_class`. Dans d'autres environnements, vous devriez vous assurer que la transaction est terminée à l'appel de `Session.close()` ou `Session.disconnect()`.

20.2.5. Stratégie : transactionnelle

La stratégie de cache `transactional` supporte un cache complètement transactionnel comme, par exemple, JBoss TreeCache. Un tel cache ne peut être utilisé que dans un environnement JTA et vous devez spécifier `hibernate.transaction.manager_lookup_class`.

20.2.6. Support de stratégie de concurrence du fournisseur-cache



Important

Aucun des caches livrés ne supporte toutes les stratégies de concurrence. Le tableau suivant montre quels caches sont compatibles avec quelles stratégies de concurrence.

Aucun des caches livrés ne supporte toutes les stratégies de concurrence. Le tableau suivant montre quels caches sont compatibles avec quelles stratégies de concurrence.

Tableau 20.2. Support de stratégie de concurrence du cache

Cache	read-only (lecture seule)	nonstrict-read-write (lecture-écriture non stricte)	read-write (lecture-écriture)	transactional (transactionnel)
Table de hachage (ne pas utiliser en production)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss Cache 1.x	yes	yes		
JBoss Cache 2	yes	yes		

20.3. Gérer les caches

A chaque fois que vous passez un objet à la méthode `save()`, `update()` ou `saveOrUpdate()` et à chaque fois que vous récupérez un objet avec `load()`, `get()`, `list()`, `iterate()` ou `scroll()`, cet objet est ajouté au cache interne de la `Session`.

Lorsqu'il y a un appel à la méthode `flush()`, l'état de cet objet va être synchronisé avec la base de données. Si vous ne voulez pas que cette synchronisation ait lieu ou si vous traitez un grand nombre d'objets et que vous avez besoin de gérer la mémoire de manière efficace, vous pouvez utiliser la méthode `evict()` pour supprimer l'objet et ses collections dépendantes du cache de premier niveau de la session.

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
```



```
doSomethingWithACat(cat);
sess.evict(cat);
}
```

La `Session` fournit également une méthode `contains()` pour déterminer si une instance appartient au cache de la session.

Pour retirer tous les objets du cache session, appelez `Session.clear()`

Pour le cache de second niveau, il existe des méthodes définies dans `SessionFactory` pour retirer du cache d'une instance, de la classe entière, d'une instance de collection ou du rôle entier d'une collection.

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class); //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

Le `CacheMode` contrôle la manière dont une session particulière interagit avec le cache de second niveau :

- `CacheMode.NORMAL` - lit et écrit les articles dans le cache de second niveau
- `CacheMode.GET` - lit les articles du cache de second niveau mais ne les écrit pas sauf dans le cas d'une mise à jour des données
- `CacheMode.PUT` - écrit les articles dans le cache de second niveau mais ne les lit pas dans le cache de second niveau
- `CacheMode.REFRESH` - écrit les articles dans le cache de second niveau mais ne les lit pas dans le cache de second niveau, outrepassant l'effet de `hibernate.cache.use_minimal_puts`, en forçant un rafraîchissement du cache de second niveau pour chaque article lu dans la base de données.

Pour parcourir le contenu du cache de second niveau ou la région du cache dédiée aux requêtes, vous pouvez utiliser l'API `Statistics` :

```
Map cacheEntries = sessionFactory.getStatistics()
    .getSecondLevelCacheStatistics(regionName)
    .getEntries();
```

Vous devez pour cela activer les statistiques et optionnellement forcer Hibernate à conserver les entrées dans le cache sous un format plus compréhensible pour l'utilisateur :

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

20.4. Le cache de requêtes

Query result sets can also be cached. This is only useful for queries that are run frequently with the same parameters.

20.4.1. Enabling query caching

Caching of query results introduces some overhead in terms of your applications normal transactional processing. For example, if you cache results of a query against Person Hibernate will need to keep track of when those results should be invalidated because changes have been committed against Person. That, coupled with the fact that most applications simply gain no benefit from caching query results, leads Hibernate to disable caching of query results by default. To use query caching, you will first need to enable the query cache:

```
hibernate.cache.use_query_cache true
```

This setting creates two new cache regions:

- `org.hibernate.cache.StandardQueryCache`, holding the cached query results
- `org.hibernate.cache.UpdateTimestampsCache`, holding timestamps of the most recent updates to queryable tables. These are used to validate the results as they are served from the query cache.



Important

If you configure your underlying cache implementation to use expiry or timeouts is very important that the cache timeout of the underlying cache region for the `UpdateTimestampsCache` be set to a higher value than the timeouts of any of the query caches. In fact, we recommend that the `UpdateTimestampsCache` region not be configured for expiry at all. Note, in particular, that an LRU cache expiry policy is never appropriate.

As mentioned above, most queries do not benefit from caching or their results. So by default, individual queries are not cached even after enabling query caching. To enable results caching for a particular query, call `org.hibernate.Query.setCacheable(true)`. This call allows the query to look for existing cache results or add its results to the cache when it is executed.



Note

The query cache does not cache the state of the actual entities in the cache; it caches only identifier values and results of value type. For this reason, the query cache should always be used in conjunction with the second-level cache for those

entities expected to be cached as part of a query result cache (just as with collection caching).

20.4.2. Query cache regions

Si vous avez besoin de contrôler finement les délais d'expiration du cache, vous pouvez spécifier une région de cache nommée pour une requête particulière en appelant `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

If you want to force the query cache to refresh one of its regions (disregard any cached results it finds there) you can use `org.hibernate.Query.setCacheMode(CacheMode.REFRESH)`. In conjunction with the region you have defined for the given query, Hibernate will selectively force the results cached in that particular region to be refreshed. This is particularly useful in cases where underlying data may have been updated via a separate process and is a far more efficient alternative to bulk eviction of the region via `org.hibernate.SessionFactory.evictQueries()`.

20.5. Comprendre les performances des collections

Dans les sections précédentes, nous avons couvert les collections et leurs applications. Dans cette section, nous allons explorer de nouveaux problèmes en rapport aux collections en cours d'exécution.

20.5.1. Taxinomie

Hibernate définit trois types de collections de base :

- les collections de valeurs
- Association un-à-plusieurs
- les associations plusieurs-à-plusieurs

Cette classification distingue les différentes relations entre les tables et les clés étrangères mais ne n'en dit pas suffisamment sur le modèle relationnel. Pour comprendre parfaitement la structure relationnelle et les caractéristiques des performances, nous devons considérer la structure de la clé primaire qui est utilisée par Hibernate pour mettre à jour ou supprimer les lignes des collections. Cela nous amène aux classifications suivantes :

- collections indexées

- ensembles (sets)
- sacs (bags)

Toutes les collections indexées (maps, lists, arrays) ont une clé primaire constituée des colonnes clés (`<key>`) et `<index>`. Avec ce type de clé primaire, la mise à jour de collection est en général très performante - la clé primaire peut être indexées efficacement et une ligne particulière peut être localisée efficacement lorsque Hibernate essaie de la mettre à jour ou de la supprimer.

Les ensembles ont une clé primaire composée de `<key>` et des colonnes représentant l'élément. Elle est donc moins efficace pour certains types d'éléments de collection, en particulier les éléments composites, les textes volumineux ou les champs binaires ; la base de données peut ne pas être capable d'indexer aussi efficacement une clé primaire aussi complexe. Cependant, pour les associations un-à-plusieurs ou plusieurs-à-plusieurs, en particulier lorsqu'on utilise des entités ayant des identifiants techniques, il est probable que cela soit aussi efficace (note : si vous voulez que `SchemaExport` crée effectivement la clé primaire d'un `<set>` pour vous, vous devez déclarer toutes les colonnes avec `not-null="true"`).

Le mappage à l'aide de `<idbag>` définit une clé de substitution ce qui leur permet d'être très efficaces lors de la mise à jour. En fait il s'agit du meilleur cas de mise à jour d'une collection.

Le pire cas intervient pour les sacs. Dans la mesure où un sac permet la duplication des valeurs d'éléments et n'a pas de colonne d'index, aucune clé primaire ne peut être définie. Hibernate n'a aucun moyen de distinguer entre les lignes dupliquées. Hibernate résout ce problème en supprimant complètement (via un simple `DELETE`), puis en recréant la collection chaque fois qu'elle change. Ce qui peut être très inefficace.

Notez que pour une relation un-à-plusieurs, la "clé primaire" peut ne pas être la clé primaire de la table en base de données - mais même dans ce cas, la classification ci-dessus reste utile (Elle explique comment Hibernate localise les lignes individuelles de la collection).

20.5.2. Les lists, les maps, les idbags et les ensembles sont les collections les plus efficaces pour la mise à jour

La discussion précédente montre clairement que les collections indexées et (la plupart du temps) les ensembles, permettent de réaliser le plus efficacement les opérations d'ajout, de suppression ou mise à jour d'éléments.

Les collections indexées ont un avantage sur les ensembles, dans le cadre des associations plusieurs-à-plusieurs ou de collections de valeurs. À cause de la structure inhérente d'un `Set`, Hibernate n'effectue jamais de ligne `UPDATE` quand un élément est modifié. Les modifications apportées à un `Set` se font via un `INSERT` et `DELETE` (de chaque ligne). Une fois de plus, ce cas ne s'applique pas aux associations un-à-plusieurs.

Après s'être rappelé que les tableaux ne peuvent pas être chargés en différé, nous pouvons conclure que les lists, les maps et les idbags sont les types de collections (non inversées) les plus performants, avec les ensembles pas loin derrière. Les ensembles sont le type de collection le

plus courant dans les applications Hibernate. Cela vient du fait que la sémantique des ensembles est la plus naturelle dans le modèle relationnel.

Cependant, dans des modèles objet bien conçus avec Hibernate, on constate que la plupart des collections sont en fait des associations un-à-plusieurs avec `inverse="true"`. Pour ces associations, les mises à jour sont gérées au niveau de l'association "plusieurs-à-un" et les considérations de performance de mise à jour des collections ne s'appliquent tout simplement pas dans ces cas-là.

20.5.3. Les sacs et les listes sont les plus efficaces pour les collections inverses

Avant que vous n'oubliez les sacs pour toujours, il y a un cas précis où les sacs (et les listes) sont bien plus performants que les ensembles. Pour une collection marquée comme `inverse="true"` (le choix le plus courant pour une relation un-à-plusieurs bidirectionnelle), nous pouvons ajouter des éléments à un sac ou une liste sans avoir besoin de l'initialiser (charger) les éléments du sac! Ceci parce que `Collection.add()` ou `Collection.addAll()` doit toujours retourner vrai pour un sac ou une `List` (contrairement au `Set`). Cela peut rendre le code suivant beaucoup plus rapide :

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c); //no need to fetch the collection!
sess.flush();
```

20.5.4. Suppression en un coup

Parfois, effacer les éléments d'une collection un par un peut être extrêmement inefficace. Hibernate n'est pas totalement stupide, il sait qu'il ne faut pas le faire dans le cas d'une collection complètement vidée (lorsque vous appelez `list.clear()`, par exemple). Dans ce cas, Hibernate fera un simple `DELETE` et le travail est fait !

Supposons que nous ajoutons un élément unique dans une collection de taille vingt et que nous enlevions ensuite deux éléments. Hibernate effectuera un `INSERT` puis deux `DELETE` (à moins que la collection ne soit un sac). Cela est préférable.

Cependant, supposons que nous enlevions dix-huit éléments, laissant ainsi deux éléments, puis que nous ajoutons trois nouveaux éléments. Il y a deux moyens de procéder.

- effacer dix-huit lignes une à une puis en insérer trois
- effacer la totalité de la collection (en un SQL `DELETE`) puis insérer les cinq éléments restant (un à un)

Hibernate n'est pas assez intelligent pour savoir que, dans ce cas, la seconde option est plus rapide (Il vaut mieux que Hibernate ne soit pas trop intelligent ; un tel comportement pourrait rendre l'utilisation de triggers de bases de données plutôt aléatoire, etc...).

Heureusement, vous pouvez forcer ce comportement (c'est-à-dire la deuxième stratégie) à tout moment en libérant (c'est-à-dire en déréférençant) la collection initiale et en retournant une collection nouvellement instanciée avec tous les éléments restants.

Bien sûr, la suppression en un coup ne s'applique pas pour les collections qui sont mappées avec `inverse="true"`.

20.6. Moniteur de performance

L'optimisation n'est pas d'un grand intérêt sans le suivi et l'accès aux données de performance. Hibernate fournit toute une panoplie de rapport sur ses opérations internes. Les statistiques dans Hibernate sont fournies par `SessionFactory`.

20.6.1. Suivi d'une SessionFactory

Vous pouvez accéder aux métriques d'une `SessionFactory` de deux manières. La première option est d'appeler `sessionFactory.getStatistics()` et de lire ou d'afficher les `Statistics` vous-même.

Hibernate peut également utiliser JMX pour publier les métriques si vous activez le `MBean StatisticsService`. Vous pouvez activer un seul `MBean` pour toutes vos `SessionFactory` ou un par fabrique. Voici un code qui montre un exemple de configuration minimaliste :

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

Vous pouvez (dés)activer le suivi pour une `SessionFactory` :

- au moment de la configuration en mettant `hibernate.generate_statistics` à `false`
- à chaud avec `sf.getStatistics().setStatisticsEnabled(true)` ou `hibernateStatsBean.setStatisticsEnabled(true)`

Les statistiques peuvent être remises à zéro de manière programmatique à l'aide de la méthode `clear()`. Un résumé peut être envoyé à un logger (niveau info) à l'aide de la méthode `logSummary()`.

20.6.2. Métriques

Hibernate fournit plusieurs métriques, qui vont des informations très basiques aux informations très spécialisées qui ne sont appropriées que dans certains scénarios. Tous les compteurs accessibles sont décrits dans l'API de l'interface `Statistics` dans trois catégories :

- Les métriques relatives à l'usage général de la `Session` comme le nombre de sessions ouvertes, le nombre de connexions JDBC récupérées, etc...
- Les métriques relatives aux entités, collections, requêtes et caches dans leur ensemble (métriques globales aka).
- Les métriques détaillées relatives à une entité, une collection, une requête ou une région de cache particulière.

Par exemple, vous pouvez vérifier les hit, miss du cache ainsi que le taux d'éléments manquants et de mise à jour des entités, collections et requêtes et le temps moyen que met une requête. Il faut faire attention au fait que le nombre de millisecondes est sujet à approximation en Java. Hibernate est lié à la précision de la machine virtuelle, sur certaines plateformes, cela n'offre qu'une précision de l'ordre de 10 secondes.

Des accesseurs simples sont utilisés pour accéder aux métriques globales (c'est-à-dire, celles qui ne sont pas liées à une entité, collection ou région de cache particulière). Vous pouvez accéder aux métriques d'une entité, collection, région de cache particulière à l'aide de son nom et à l'aide de sa représentation HQL ou SQL pour une requête. Référez vous à la javadoc des APIs `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, et `QueryStatistics` pour plus d'informations. Le code ci-dessous montre un exemple simple :

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
    queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
    stats.getEntityStatistics( Cat.class.getName() );
long changes =
    entityStats.getInsertCount()
    + entityStats.getUpdateCount()
    + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + " times ");
```

Pour travailler sur toutes les entités, collections, requêtes et régions de cache, vous pouvez récupérer la liste des noms des entités, collections, requêtes et régions de cache avec les méthodes suivantes : `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, et `getSecondLevelCacheRegionNames()`.

Guide de la boîte à outils

Des outils en ligne de commande, des plugins Eclipse ainsi que des tâches Ant permettent de gérer le développement complet de projets à travers Hibernate.

Les *outils Hibernate* actuels incluent des plugins pour l'IDE Eclipse ainsi que des tâches Ant pour l'ingénierie inverse de bases de données existantes :

- *Mapping Editor* : un éditeur pour les fichiers de mappage XML Hibernate, supportant l'auto-finalisation et la mise en valeur de la syntaxe. Il supporte aussi la sémantique d'auto-finalisation pour les noms de classes et les noms de propriété/champs, le rendant beaucoup plus polyvalent qu'un éditeur XML ordinaire.
- *Console* : la console est une nouvelle vue d'Eclipse. En plus de la vue d'ensemble arborescente de vos configurations de console, vous obtenez aussi une vue interactive de vos classes persistantes et de leurs relations. La console vous permet d'exécuter des requête HQL dans votre base de données et de parcourir les résultats directement dans Eclipse.
- *Development Wizards* : plusieurs assistants sont fournis avec les outils de Hibernate pour Eclipse ; vous pouvez utiliser un assistant pour générer rapidement les fichiers de configuration Hibernate (cfg.xml), ou vous pouvez même complètement générer les fichiers de mappage Hibernate et les sources des POJOs à partir d'un schéma de base de données existant. L'assistant d'ingénierie inverse supporte les modèles utilisateur.
-

Veuillez-vous référer au paquetage *Outils Hibernate* et à sa documentation pour plus d'informations.

Cependant, le paquetage principal de Hibernate arrive avec un ensemble d'outils intégrés (il peut même être utilisé de "l'intérieur" de Hibernate à la volée) : *SchemaExport* aussi connu comme hbm2ddl.

21.1. Génération automatique du schéma

La DDL peut être générée à partir de vos fichiers de mappage par un utilitaire Hibernate. Le schéma généré inclut les contraintes d'intégrité référentielle (clefs primaires et étrangères) pour les tables d'entités et de collections. Les tables et les séquences sont aussi créées pour les générateurs d'identifiants mappés.

Vous devez spécifier un `Dialect SQL` via la propriété `hibernate.dialect` lors de l'utilisation de cet outil, puisque la DDL est fortement dépendante du vendeur spécifique.

D'abord, personnalisez vos fichiers de mappage pour améliorer le schéma généré.

21.1.1. Personnaliser le schéma

Plusieurs éléments du mappage Hibernate définissent des attributs optionnels nommés `length`, `precision` et `scale`. Vous pouvez paramétrer la taille, la précision, et l'échelle d'une colonne avec cet attribut.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Certaines balises acceptent aussi un attribut `not-null` utilisé pour générer les contraintes de colonnes `NOT NULL` et un attribut `unique` pour générer une contrainte `UNIQUE` de colonnes de table.

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

Un attribut `unique-key` peut être utilisé pour grouper les colonnes en une seule contrainte d'unicité. Actuellement, la valeur spécifiée par l'attribut `unique-key` n'est pas utilisée pour nommer la contrainte dans la DDL générée, elle sert juste à grouper les colonnes dans le fichier de mappage.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployeeId"/>
```

Un attribut `index` indique le nom d'un index qui sera créé en utilisant la ou les colonnes mappées. Plusieurs colonnes peuvent être groupées dans un même index, en spécifiant le même nom d'index.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

Un attribut `foreign-key` peut être utilisé pour surcharger le nom des clés étrangères générées.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Plusieurs éléments de mappage acceptent aussi un élément fils `<column>`. Ceci est particulièrement utile pour les type multi-colonnes :

```
<property name="name" type="my.customtypes.Name"/>
  <column name="last" not-null="true" index="bar_idx" length="30"/>
  <column name="first" not-null="true" index="bar_idx" length="20"/>
  <column name="initial"/>
</property>
>
```

L'attribut `default` vous laisse spécifier une valeur par défaut pour une colonne. Vous devez assigner la même valeur à la propriété mappée avant de sauvegarder une nouvelle instance de la classe mappée.

```
<property name="credits" type="integer" insert="false">
  <column name="credits" default="10"/>
</property>
>
```

```
<version name="version" type="integer" insert="false">
  <column name="version" default="0"/>
</property>
>
```

L'attribut `sql-type` permet à l'utilisateur de surcharger le mappage par défaut d'un type Hibernate vers un type de données SQL.

```
<property name="balance" type="float">
  <column name="balance" sql-type="decimal(13,3)"/>
</property>
>
```

L'attribut `check` permet de spécifier une contrainte de vérification.

```
<property name="foo" type="integer">
  <column name="foo" check="foo
> 10"/>
</property>
>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
  ...
  <property name="bar" type="float"/>
```

```
</class>
>
```

Le tableau suivant dresse la liste des attributs en option.

Tableau 21.1. Résumé

Attribut	Valeurs	Interprétation
length	numérique	taille d'une colonne
precision	numérique	précision décimale de la colonne
scale	numérique	échelle décimale de la colonne
not-null	true false	spécifie que la colonne doit être non-nulle
unique	true false	spécifie que la colonne doit avoir une contrainte d'unicité
index	index_name	spécifie le nom d'un index (multi-colonnes)
unique-key	unique_key_name	spécifie le nom d'une contrainte d'unicité multi-colonnes
foreign-key	foreign_key_name	spécifie le nom de la contrainte de clé étrangère générée par une association, pour un élément de mappage <one-to-one>, <many-to-one>, <key>, ou <many-to-many>. Notez que les côtés <code>inverse="true"</code> ne seront pas pris en considération par le <code>SchemaExport</code> .
sql-type	SQL column type	surcharge le type par défaut (attribut de l'élément <column> uniquement)
default	Expression SQL	spécifie une valeur par défaut pour la colonne
check	Expression SQL	crée une contrainte de vérification sur la table ou la colonne

L'élément <comment> vous permet de spécifier des commentaires pour le schéma généré.

```
<class name="Customer" table="CurCust">
  <comment
>Current customers only</comment>
  ...
</class>
>
```

```
<property name="balance">
  <column name="bal">
    <comment
>Balance in USD</comment>
  </column>
```

```
</property>
>
```

Ceci a pour résultat une expression `comment on table` ou `comment on column` dans la DDL générée (là où elle est supportée).

21.1.2. Exécuter l'outil

L'outil `SchemaExport` génère un script DDL vers la sortie standard et/ou exécute les ordres DDL.

Le tableau suivant affiche les options de ligne de commande du `SchemaExport`

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaExport options
mapping_files
```

Tableau 21.2. `SchemaExport` Options de la ligne de commande

Option	Description
<code>--quiet</code>	ne pas écrire le script vers la sortie standard stdout
<code>--drop</code>	supprime uniquement les tables
<code>--create</code>	ne crée que les tables
<code>--text</code>	n'exporte pas vers la base de données
<code>--output=my_schema.ddl</code>	écrit le script ddl vers un fichier
<code>--naming=eg.MyNamingStrategy</code>	sélectionne une <code>NamingStrategy</code>
<code>--config=hibernate.cfg.xml</code>	lit la configuration Hibernate à partir d'un fichier XML
<code>--properties=hibernate.properties</code>	lit les propriétés de la base de données à partir d'un fichier
<code>--format</code>	formate proprement le SQL généré dans le script
<code>--delimiter=;</code>	paramètre un délimiteur de fin de ligne pour le script

Vous pouvez même intégrer `SchemaExport` dans votre application :

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

21.1.3. Propriétés

Les propriétés de la base de données peuvent être spécifiées :

- comme propriétés système avec `-D<property>`
- dans `hibernate.properties`
- dans un fichier de propriétés déclaré avec `--properties`

Les propriétés nécessaires sont :

Tableau 21.3. Les propriétés de connexion SchemaExport

Nom de la propriété	Description
hibernate.connection.driver_class	classe du driver JDBC
hibernate.connection.url	URL JDBC
hibernate.connection.username	utilisateur de la base de données
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.dialect	dialecte

21.1.4. Utiliser Ant

Vous pouvez appeler SchemaExport depuis votre script de construction Ant :

```
<target name="schemaexport">
  <taskdef name="schemaexport"
    classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
    classpathref="class.path" />

  <schemaexport
    properties="hibernate.properties"
    quiet="no"
    text="no"
    drop="no"
    delimiter=";"
    output="schema-export.sql">
    <fileset dir="src">
      <include name="**/*.hbm.xml" />
    </fileset>
  </schemaexport>
</target>
>
```

21.1.5. Mises à jour incrémentales du schéma

L'outil SchemaUpdate mettra à jour un schéma existant en effectuant les changements par "incrément". Notez que SchemaUpdate dépend fortement de l'API des métadonnées JDBC, par conséquent il ne fonctionne pas avec tous les drivers JDBC.

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaUpdate options
mapping_files
```

Tableau 21.4. SchemaUpdate Options de ligne de commande

Option	Description
--quiet	ne pas écrire le script vers la sortie standard stdout
--text	ne pas exporter vers la base de données

Option	Description
--naming=eg.MyNamingStrategy	sélectionne une NamingStrategy
-- properties=hibernate.properties	lit les propriétés de la base de données à partir d'un fichier
--config=hibernate.cfg.xml	spécifier un fichier .cfg.xml

Vous pouvez intégrer SchemaUpdate dans votre application :

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

21.1.6. Utiliser Ant pour des mises à jour de schéma par incrément

Vous pouvez appeler SchemaUpdate depuis le script Ant :

```
<target name="schemaupdate">
  <taskdef name="schemaupdate"
    classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
    classpathref="class.path"/>

  <schemaupdate
    properties="hibernate.properties"
    quiet="no">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemaupdate>
</target>
>
```

21.1.7. Validation du schéma

L'outil SchemaValidator confirmera que le schéma existant correspond à vos documents de mappage. Notez que le SchemaValidator dépend de l'API des métadonnées de JDBC, il ne fonctionne donc pas avec tous les drivers JDBC. Cet outil est extrêmement utile pour les tests.

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaValidator options
mapping_files
```

Le tableau suivant affiche les options de ligne de commande du SchemaValidator

Tableau 21.5. SchemaValidator Options de ligne de commande

Option	Description
--naming=eg.MyNamingStrategy	sélectionne une NamingStrategy

Option	Description
-- properties=hibernate.properties	lit les propriétés de la base de données à partir d'un fichier
--config=hibernate.cfg.xml	spécifier un fichier .cfg.xml

Vous pouvez inclure `SchemaValidator` dans votre application :

```
Configuration cfg = ....;  
new SchemaValidator(cfg).validate();
```

21.1.8. Utiliser Ant pour la validation du Schéma

Vous pouvez appeler `SchemaValidator` depuis le script Ant:

```
<target name="schemavalidate">  
  <taskdef name="schemavalidator"  
    classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"  
    classpathref="class.path" />  
  
  <schemavalidator  
    properties="hibernate.properties">  
    <fileset dir="src">  
      <include name="**/*.hbm.xml" />  
    </fileset>  
  </schemavalidator>  
</target>  
>
```


Exemple : père/fils

L'une des premières choses que les nouveaux utilisateurs essaient de faire avec Hibernate est de modéliser une relation père/fils. Il y a deux approches différentes pour cela. Pour un certain nombre de raisons, la méthode la plus courante, en particulier pour les nouveaux utilisateurs, est de modéliser les deux relations `Père` et `Fils` comme des classes entités liées par une association `<one-to-many>` du `Père` vers le `Fils` (l'autre approche est de déclarer le `Fils` comme un `<composite-element>`). On constate que la sémantique par défaut de l'association un-à-plusieurs (dans Hibernate) est bien moins proche du sens habituel d'une relation père/fils que celle d'un mappage d'élément composite. Nous allons vous expliquer comment utiliser une association *un-à-plusieurs bidirectionnelle avec cascade* afin de modéliser efficacement et élégamment une relation père/fils.

22.1. Une note à propos des collections

Les collections Hibernate sont considérées comme étant une partie logique de leur entité propriétaire, jamais des entités qu'elle contient. C'est une distinction cruciale ! Les conséquences sont les suivantes :

- Quand nous ajoutons / retirons un objet d'une collection, le numéro de version du propriétaire de la collection est incrémenté.
- Si un objet qui a été enlevé d'une collection est une instance de type valeur (par ex : élément composite), cet objet cessera d'être persistant et son état sera complètement effacé de la base de données. Par ailleurs, ajouter une instance de type valeur dans une collection entraînera que son état sera immédiatement persistant.
- Si une entité est enlevée d'une collection (association un-à-plusieurs ou plusieurs-à-plusieurs), elle ne sera pas effacée par défaut. Ce comportement est complètement logique - une modification de l'un des états internes d'une autre entité ne doit pas causer la disparition de l'entité associée. De même, l'ajout d'une entité dans une collection n'engendre pas, par défaut, la persistance de cette entité.

Le comportement par défaut est donc que l'ajout d'une entité dans une collection crée simplement le lien entre les deux entités, alors qu'effacer une entité supprime ce lien. C'est le comportement le plus approprié dans la plupart des cas. Ce comportement n'est cependant pas approprié lorsque la vie du fils est liée au cycle de vie du père.

22.2. Un-à-plusieurs bidirectionnel

Supposons que nous ayons une simple association `<one-to-many>` de `Parent` à `Child`.

```
<set name="children">
  <key column="parent_id"/>
```

```
<one-to-many class="Child"/>
</set>
>
```

Si nous exécutons le code suivant :

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate exécuterait deux ordres SQL :

- un INSERT pour créer l'enregistrement pour `c`
- un UPDATE pour créer le lien de `p` vers `c`

Ceci est non seulement inefficace, mais viole aussi toute contrainte NOT NULL sur la colonne `parent_id`. Nous pouvons réparer la contrainte de nullité en spécifiant `not-null="true"` dans le mappage de la collection :

```
<set name="children">
  <key column="parent_id" not-null="true"/>
  <one-to-many class="Child"/>
</set>
>
```

Cependant ce n'est pas la solution recommandée.

La cause sous jacente à ce comportement est que le lien (la clé étrangère `parent_id`) de `p` vers `c` n'est pas considérée comme faisant partie de l'état de l'objet `Child` et n'est donc pas créé par l'INSERT. La solution est donc que ce lien fasse partie du mappage de `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

Nous avons aussi besoin d'ajouter la propriété `parent` dans la classe `Child`.

Maintenant que l'état du lien est géré par l'entité `Child`, nous spécifions à la collection de ne pas mettre à jour le lien. Nous utilisons l'attribut `inverse` pour faire cela :

```
<set name="children" inverse="true">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

>

Le code suivant serait utilisé pour ajouter un nouveau `Child` :

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

Maintenant, seul un SQL `INSERT` est nécessaire.

Pour alléger encore un peu les choses, nous créerons une méthode `addChild()` de `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Le code d'ajout d'un `Child` serait alors :

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

22.3. Cycle de vie en cascade

L'appel explicite de `save()` est un peu fastidieux. Nous pouvons simplifier cela en utilisant les cascades.

```
<set name="children" inverse="true" cascade="all">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
>
```

Cela simplifie le code précédent en :

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
```

```
session.flush();
```

De la même manière, nous n'avons pas à itérer sur les fils lorsque nous sauvons ou effaçons un `Parent`. Le code suivant efface `p` et tous ses fils de la base de données.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Par contre, ce code :

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

n'effacera pas `c` de la base de données, il enlèvera seulement le lien vers `p` (et causera une violation de contrainte `NOT NULL`, dans ce cas). Vous devez explicitement utiliser `delete()` sur `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

Dans notre cas, un `Child` ne peut pas vraiment exister sans son père. Si nous effaçons un `Child` de la collection, nous voulons vraiment qu'il soit effacé. Pour cela, nous devons utiliser `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
>
```

À noter : même si le mappage de la collection spécifie `inverse="true"`, les cascades sont toujours assurées par l'itération sur les éléments de la collection. Donc, si vous avez besoin qu'un objet soit enregistré, effacé ou mis à jour par cascade, vous devez l'ajouter dans la collection. Il ne suffit pas d'appeler explicitement `setParent()`.

22.4. Cascades et `unsaved-value` (valeurs non sauvegardées)

Suppose we loaded up a `Parent` in one `Session`, made some changes in a UI action and wanted to persist these changes in a new session by calling `update()`. The `Parent` will contain a collection of children and, since the cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both `Parent` and `Child` have generated identifier properties of type `Long`. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See [Section 10.7, « Détection automatique d'un état »](#).) In *Hibernate3*, it is no longer necessary to specify an `unsaved-value` explicitly.

Le code suivant mettra à jour `parent` et `child` et insérera `newChild`.

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Ceci est très bien pour des identifiants générés, mais qu'en est-il des identifiants assignés et des identifiants composés ? C'est plus difficile, puisque Hibernate ne peut pas utiliser la propriété de l'identifiant pour distinguer entre un objet nouvellement instancié (avec un identifiant assigné par l'utilisateur) et un objet chargé dans une session précédente. Dans ce cas, Hibernate utilisera soit la propriété de version ou d'horodatage, soit effectuera vraiment une requête au cache de second niveau, soit, dans le pire des cas, à la base de données, pour voir si la ligne existe.

22.5. Conclusion

Il y a quelques principes à maîtriser dans ce chapitre et tout cela peut paraître déroutant la première fois. Cependant, dans la pratique, tout fonctionne parfaitement. La plupart des applications Hibernate utilisent le modèle père / fils.

Nous avons évoqué une alternative dans le premier paragraphe. Aucun des points traités précédemment n'existe dans le cas de mappings `<composite-element>` qui possède exactement la sémantique d'une relation père / fils. Malheureusement, il y a deux grandes limitations pour les classes d'éléments composites : les éléments composites ne peuvent contenir de collections, et ils ne peuvent être les fils d'entités autres que l'unique parent.

Exemple : application Weblog

23.1. Classes persistantes

Les classes persistantes représentent un weblog, et un article posté dans un weblog. Il seront modélisés comme une relation père/fils standard, mais nous allons utiliser un sac trié au lieu d'un set :

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
}
```

```
public Calendar getDatetime() {
    return _datetime;
}
public Long getId() {
    return _id;
}
public String getText() {
    return _text;
}
public String getTitle() {
    return _title;
}
public void setBlog(Blog blog) {
    _blog = blog;
}
public void setDatetime(Calendar calendar) {
    _datetime = calendar;
}
public void setId(Long long1) {
    _id = long1;
}
public void setText(String string) {
    _text = string;
}
public void setTitle(String string) {
    _title = string;
}
}
```

23.2. Mappages Hibernate

Le mappage XML doit maintenant être relativement simple à vos yeux. Par exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
```



```

        unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
>

```

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>
    </class>
</hibernate-mapping>

```

```
</class>

</hibernate-mapping>
>
```

23.3. Code Hibernate

La classe suivante montre quelques utilisations de ces classes avec Hibernate :

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
    }
}
```

```

        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }
}

public BlogItem createBlogItem(Blog blog, String title, String text)
    throws HibernateException {

    BlogItem item = new BlogItem();
    item.setTitle(title);
    item.setText(text);
    item.setBlog(blog);
    item.setDatetime( Calendar.getInstance() );
    blog.getItems().add(item);

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(blog);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return item;
}

public BlogItem createBlogItem(Long blogid, String title, String text)
    throws HibernateException {

    BlogItem item = new BlogItem();
    item.setTitle(title);
    item.setText(text);
    item.setDatetime( Calendar.getInstance() );

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Blog blog = (Blog) session.load(Blog.class, blogid);
        item.setBlog(blog);
        blog.getItems().add(item);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {

```

```

        session.close();
    }
    return item;
}

public void updateBlogItem(BlogItem item, String text)
    throws HibernateException {

    item.setText(text);

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(item);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

public void updateBlogItem(Long itemid, String text)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
        item.setText(text);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

public List listAllBlogNamesAndItemCounts(int max)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    List result = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "select blog.id, blog.name, count(blogItem) " +
            "from Blog as blog " +
            "left outer join blog.items as blogItem " +
            "group by blog.name, blog.id " +

```

```

        "order by max(blogItem.datetime)"
    );
    q.setMaxResults(max);
    result = q.list();
    tx.commit();
}
catch (HibernateException he) {
    if (tx!=null) tx.rollback();
    throw he;
}
finally {
    session.close();
}
return result;
}

public Blog getBlogAndAllItems(Long blogid)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    Blog blog = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "left outer join fetch blog.items " +
            "where blog.id = :blogid"
        );
        q.setParameter("blogid", blogid);
        blog = (Blog) q.uniqueResult();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return blog;
}

public List listBlogsAndRecentItems() throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    List result = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "inner join blog.items as blogItem " +
            "where blogItem.datetime
> :minDate"
        );
        tx.commit();

        Calendar cal = Calendar.getInstance();
        cal.roll(Calendar.MONTH, false);

```

```
        q.setCalendar("minDate", cal);

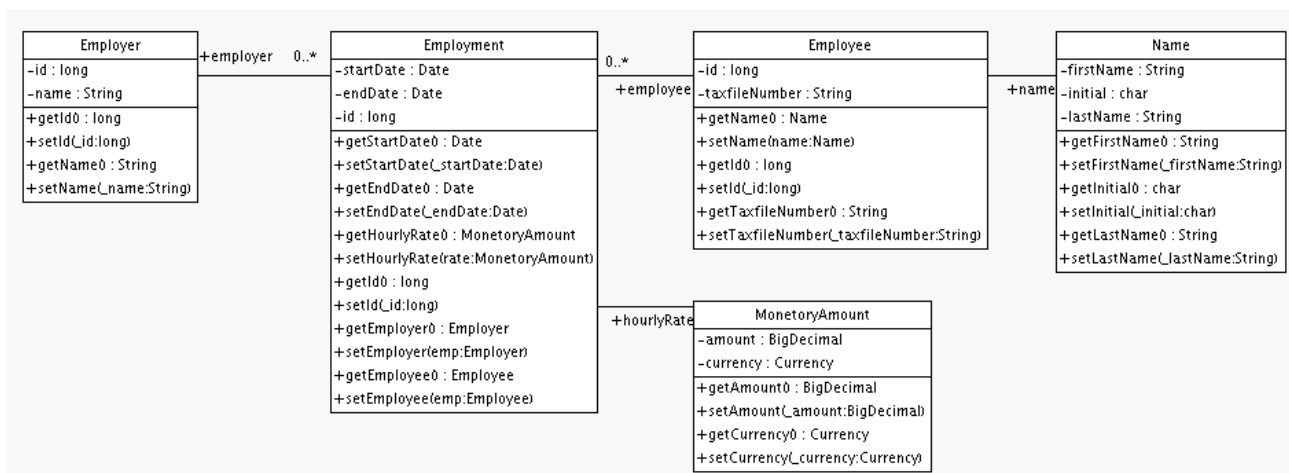
        result = q.list();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return result;
}
}
```

Exemple : quelques mappages

Ce chapitre montre quelques mappages plus complexes.

24.1. Employeur/Employé (Employer/Employee)

Le modèle suivant de relation entre `Employer` et `Employee` utilise une vraie classe entité (`Employment`) pour représenter l'association. La raison étant qu'il peut y avoir plus d'une période d'emploi pour les deux mêmes parties. Des composants sont utilisés pour modéliser les valeurs monétaires et les noms des employés.



Voici un document de mappage possible :

```

<hibernate-mapping>

  <class name="Employer" table="employers">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">
>employer_id_seq</param>
      </generator>
    </id>
    <property name="name" />
  </class>

  <class name="Employment" table="employment_periods">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">
>employment_id_seq</param>
      </generator>
    </id>
    <property name="startDate" column="start_date" />
    <property name="endDate" column="end_date" />

    <component name="hourlyRate" class="MonetaryAmount">
      <property name="amount">

```

```
        <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
    </property>
    <property name="currency" length="12"/>
</component>

<many-to-one name="employer" column="employer_id" not-null="true"/>
<many-to-one name="employee" column="employee_id" not-null="true"/>

</class>

<class name="Employee" table="employees">
    <id name="id">
        <generator class="sequence">
            <param name="sequence">
>employee_id_seq</param>
        </generator>
    </id>
    <property name="taxfileNumber"/>
    <component name="name" class="Name">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </component>
</class>

</hibernate-mapping>
>
```

Et voici le schéma des tables générées par SchemaExport.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)
```



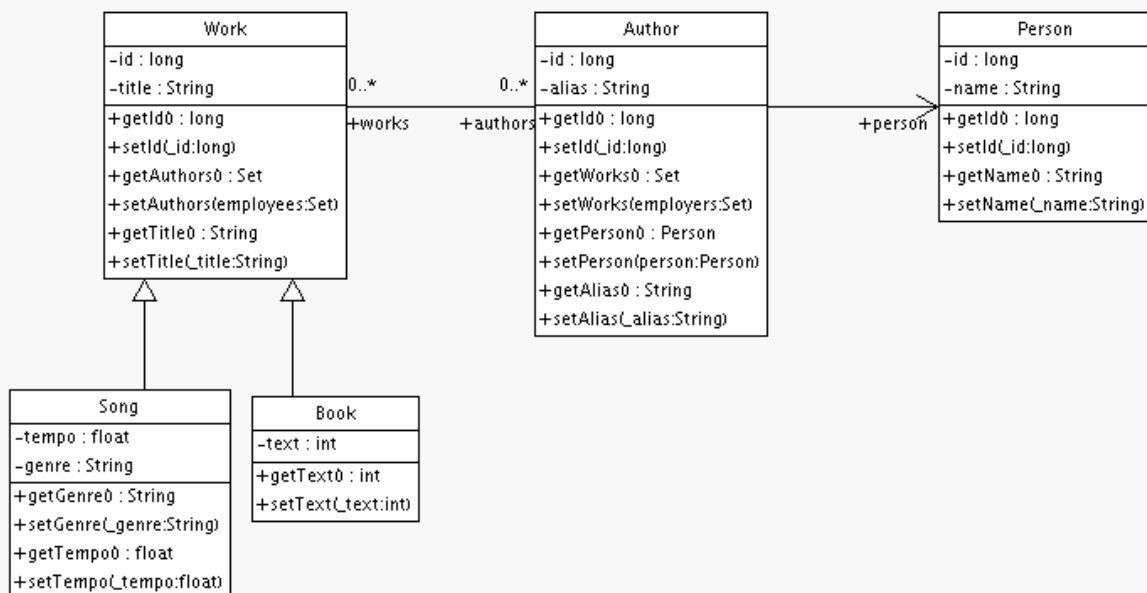
```

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq

```

24.2. Auteur/Travail

Examinons le modèle suivant de la relation entre `Work`, `Author` et `Person`. Nous représentons la relation entre `Work` et `Author` comme une association plusieurs-à-plusieurs. Nous avons choisi de représenter la relation entre `Author` et `Person` comme une association un-à-un. Une autre possibilité aurait été que `Author` étende `Person`.



Le mappage suivant représente exactement ces relations :

```

<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>
    </class>

```

```
<subclass name="Book" discriminator-value="B">
  <property name="text"/>
</subclass>

<subclass name="Song" discriminator-value="S">
  <property name="tempo"/>
  <property name="genre"/>
</subclass>

</class>

<class name="Author" table="authors">

  <id name="id" column="id">
    <!-- The Author must have the same identifier as the Person -->
    <generator class="assigned"/>
  </id>

  <property name="alias"/>
  <one-to-one name="person" constrained="true"/>

  <set name="works" table="author_work" inverse="true">
    <key column="author_id"/>
    <many-to-many class="Work" column="work_id"/>
  </set>

</class>

<class name="Person" table="persons">
  <id name="id" column="id">
    <generator class="native"/>
  </id>
  <property name="name"/>
</class>

</hibernate-mapping>
>
```

Il y a quatre tables dans ce mappage. works, authors et persons qui contiennent respectivement les données de work, author et person. author_work est une table d'association qui lie authors à works. Voici le schéma de tables, généré par SchemaExport :

```
create table works (
  id BIGINT not null generated by default as identity,
  tempo FLOAT,
  genre VARCHAR(255),
  text INTEGER,
  title VARCHAR(255),
  type CHAR(1) not null,
  primary key (id)
)

create table author_work (
  author_id BIGINT not null,
  work_id BIGINT not null,
```

```

    primary key (work_id, author_id)
  )

create table authors (
  id BIGINT not null generated by default as identity,
  alias VARCHAR(255),
  primary key (id)
)

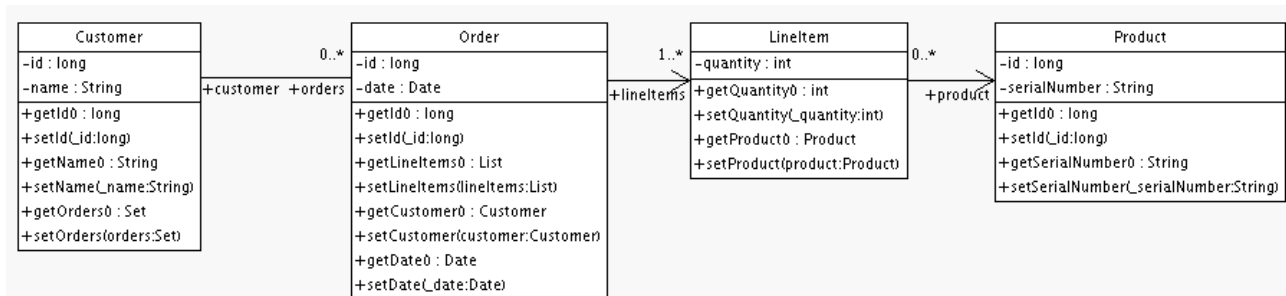
create table persons (
  id BIGINT not null generated by default as identity,
  name VARCHAR(255),
  primary key (id)
)

alter table authors
  add constraint authorsFK0 foreign key (id) references persons
alter table author_work
  add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
  add constraint author_workFK1 foreign key (work_id) references works

```

24.3. Client/Commande/Produit

Imaginons maintenant le modèle de relations entre `Customer`, `Order`, `LineItem` et `Product`. Il y a une association un-à-plusieurs entre `Customer` et `Order`, mais comment devons nous représenter `Order` / `LineItem` / `Product` ? J'ai choisi de mapper `LineItem` comme une classe d'association représentant l'association plusieurs-à-plusieurs entre `Order` et `Product`. Dans Hibernate, on appelle cela un élément composite.



Le document de mappage :

```

<hibernate-mapping>

  <class name="Customer" table="customers">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <set name="orders" inverse="true">
      <key column="customer_id"/>
      <one-to-many class="Order"/>
    </set>
  </class>

```

```
<class name="Order" table="orders">
  <id name="id">
    <generator class="native"/>
  </id>
  <property name="date"/>
  <many-to-one name="customer" column="customer_id"/>
  <list name="lineItems" table="line_items">
    <key column="order_id"/>
    <list-index column="line_number"/>
    <composite-element class="LineItem">
      <property name="quantity"/>
      <many-to-one name="product" column="product_id"/>
    </composite-element>
  </list>
</class>

<class name="Product" table="products">
  <id name="id">
    <generator class="native"/>
  </id>
  <property name="serialNumber"/>
</class>

</hibernate-mapping>
>
```

customers, orders, line_items et products contiennent les données de customer, order, order line item et product. line_items est aussi la table d'association liant orders à products.

```
create table customers (
  id BIGINT not null generated by default as identity,
  name VARCHAR(255),
  primary key (id)
)

create table orders (
  id BIGINT not null generated by default as identity,
  customer_id BIGINT,
  date TIMESTAMP,
  primary key (id)
)

create table line_items (
  line_number INTEGER not null,
  order_id BIGINT not null,
  product_id BIGINT,
  quantity INTEGER,
  primary key (order_id, line_number)
)

create table products (
  id BIGINT not null generated by default as identity,
  serialNumber VARCHAR(255),
  primary key (id)
)
```

```

alter table orders
  add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
  add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
  add constraint line_itemsFK1 foreign key (order_id) references orders

```

24.4. Divers exemples de mappages

Ces exemples sont tous pris de la suite de tests de Hibernate. Vous en trouverez beaucoup d'autres. Regardez dans le dossier `test` de la distribution Hibernate.

24.4.1. "Typed" association un-à-un

```

<class name="Person">
  <id name="name" />
  <one-to-one name="address"
    cascade="all">
    <formula
>name</formula>
    <formula
>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
      cascade="all">
      <formula
>name</formula>
      <formula
>'MAILING'</formula>
    </one-to-one>
  </class>

<class name="Address" batch-size="2"
  check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
  <composite-id>
    <key-many-to-one name="person"
      column="personName" />
    <key-property name="type"
      column="addressType" />
  </composite-id>
  <property name="street" type="text" />
  <property name="state" />
  <property name="zip" />
</class>
>

```

24.4.2. Exemple de clef composée

```

<class name="Customer">

```

```
<id name="customerId"
    length="10">
    <generator class="assigned"/>
</id>

<property name="name" not-null="true" length="100"/>
<property name="address" not-null="true" length="200"/>

<list name="orders"
    inverse="true"
    cascade="save-update">
    <key column="customerId"/>
    <index column="orderNumber"/>
    <one-to-many class="Order"/>
</list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
        class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
        type="calendar_date"
        not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
              from LineItem li, Product p
              where li.productId = p.productId
                  and li.customerId = customerId
                  and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
        column="customerId"
        insert="false"
        update="false"
        not-null="true"/>

    <bag name="lineItems"
        fetch="join"
        inverse="true"
        cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>
```

```
</class>

<class name="LineItem">

  <composite-id name="id"
    class="LineItem$Id">
    <key-property name="customerId" length="10"/>
    <key-property name="orderNumber"/>
    <key-property name="productId" length="10"/>
  </composite-id>

  <property name="quantity"/>

  <many-to-one name="order"
    insert="false"
    update="false"
    not-null="true">
    <column name="customerId"/>
    <column name="orderNumber"/>
  </many-to-one>

  <many-to-one name="product"
    insert="false"
    update="false"
    not-null="true"
    column="productId"/>

</class>

<class name="Product">
  <synchronize table="LineItem"/>

  <id name="productId"
    length="10">
    <generator class="assigned"/>
  </id>

  <property name="description"
    not-null="true"
    length="200"/>
  <property name="price" length="3"/>
  <property name="numberAvailable"/>

  <property name="numberOrdered">
    <formula>
      ( select sum(li.quantity)
        from LineItem li
        where li.productId = productId )
    </formula>
  </property>

</class>
>
```

24.4.3. Plusieurs-à-plusieurs avec un attribut de clef composée partagée

```
<class name="User" table="`User`">
  <composite-id>
    <key-property name="name" />
    <key-property name="org" />
  </composite-id>
  <set name="groups" table="UserGroup">
    <key>
      <column name="userName" />
      <column name="org" />
    </key>
    <many-to-many class="Group">
      <column name="groupName" />
      <formula
>org</formula>
    </many-to-many>
  </set>
</class>

<class name="Group" table="`Group`">
  <composite-id>
    <key-property name="name" />
    <key-property name="org" />
  </composite-id>
  <property name="description" />
  <set name="users" table="UserGroup" inverse="true">
    <key>
      <column name="groupName" />
      <column name="org" />
    </key>
    <many-to-many class="User">
      <column name="userName" />
      <formula
>org</formula>
    </many-to-many>
  </set>
</class>
```

24.4.4. Contenu basé sur une discrimination

```
<class name="Person"
  discriminator-value="P">

  <id name="id"
    column="person_id"
    unsaved-value="0">
    <generator class="native" />
  </id>

  <discriminator
```



```

    type="character">
    <formula>
      case
        when title is not null then 'E'
        when salesperson is not null then 'C'
        else 'P'
      end
    </formula>
  </discriminator>

  <property name="name"
    not-null="true"
    length="80"/>

  <property name="sex"
    not-null="true"
    update="false"/>

  <component name="address">
    <property name="address"/>
    <property name="zip"/>
    <property name="country"/>
  </component>

  <subclass name="Employee"
    discriminator-value="E">
    <property name="title"
      length="20"/>
    <property name="salary"/>
    <many-to-one name="manager"/>
  </subclass>

  <subclass name="Customer"
    discriminator-value="C">
    <property name="comments"/>
    <many-to-one name="salesperson"/>
  </subclass>

</class
>

```

24.4.5. Associations sur des clés alternées

```

<class name="Person">

  <id name="id">
    <generator class="hilo"/>
  </id>

  <property name="name" length="100"/>

  <one-to-one name="address"
    property-ref="person"
    cascade="all"
    fetch="join"/>

```

```
<set name="accounts"
  inverse="true">
  <key column="userId"
    property-ref="userId"/>
  <one-to-many class="Account"/>
</set>

<property name="userId" length="8"/>

</class>

<class name="Address">

  <id name="id">
    <generator class="hilo"/>
  </id>

  <property name="address" length="300"/>
  <property name="zip" length="5"/>
  <property name="country" length="25"/>
  <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
  <id name="accountId" length="32">
    <generator class="uuid"/>
  </id>

  <many-to-one name="user"
    column="userId"
    property-ref="userId"/>

  <property name="type" not-null="true"/>

</class>
>
```

Meilleures pratiques

Découpez finement vos classes et mappez-les en utilisant `<component>` :

Utilisez une classe `Address` pour résumer `street`, `suburb`, `state`, `postcode`. Ceci permet la réutilisation du code et simplifie la maintenance.

Déclarez des propriétés d'identifiants dans les classes persistantes :

Hibernate rend les propriétés d'identifiants optionnelles. Il est recommandé de les utiliser pour de nombreuses raisons. Utilisez les identifiants comme 'synthetic' (générés, et sans connotation métier).

Identifiez les clefs naturelles :

Identifiez les clefs naturelles pour toutes les entités, et mappez-les avec `<natural-id>`. Implémentez `equals()` et `hashCode()` pour comparer les propriétés qui composent la clef naturelle.

Placez chaque mapping de classe dans son propre fichier :

N'utilisez pas un unique document de mapping. Mappez `com.eg.Foo` dans le fichier `com/eg/Foo.hbm.xml`. Cela prend tout son sens lors d'un travail en équipe.

Chargez les mappings comme des ressources :

Déployez les mappings en même temps que les classes qu'ils mappent.

Pensez à externaliser les chaînes de requêtes :

Ceci est une bonne habitude si vos requêtes appellent des fonctions SQL qui ne sont pas au standard ANSI. Cette externalisation des chaînes de requête dans les fichiers de mapping rendra votre application plus portable.

Utilisez les variables bindées.

Comme dans JDBC, remplacez toujours les valeurs non constantes par `"?"`. N'utilisez jamais la manipulation des chaînes de caractères pour lier des valeurs non constantes dans une requête ! Encore mieux, utilisez les paramètres nommés dans les requêtes.

Ne gérez pas vous-même les connexions JDBC :

Hibernate permet à l'application de gérer les connexions JDBC. Vous ne devriez gérer vos connexions qu'en dernier recours. Si vous ne pouvez pas utiliser les systèmes de connexions livrés, considérez la fourniture de votre propre implémentation de `org.hibernate.connection.ConnectionProvider`.

Considérez l'utilisation de types personnalisés :

Supposez que vous ayez un type Java, de telle bibliothèque, qui a besoin d'être persisté mais qui ne fournit pas les accesseurs nécessaires pour le mapper comme composant. Vous devriez implémenter `org.hibernate.UserType`. Cette approche évite au code de l'application, l'implémentation de transformations vers / depuis les types Hibernate.

Utilisez du JDBC pur dans les goulots d'étranglement :

In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. Do not assume, however, that JDBC is necessarily faster. Please wait until you *know* something is a bottleneck. If you need to use direct JDBC, you can open a Hibernate `Session`, wrap your JDBC operation as a `org.hibernate.jdbc.Work` object and using that JDBC connection. This way you can still use the same transaction strategy and underlying connection provider.

Comprenez le flush de `Session` :

De temps en temps la `Session` synchronise ses états persistants avec la base de données. Les performances seront affectées si ce processus arrive trop souvent. Vous pouvez parfois minimiser les flush non nécessaires en désactivant le flush automatique ou même en changeant l'ordre des requêtes et autres opérations effectuées dans une transaction particulière.

Dans une architecture à trois couches, vous pouvez utiliser des objets détachés :

Quand vous utilisez une architecture à base de servlet / session bean, vous pouvez passer des objets chargés dans le bean session vers et depuis la couche servlet / JSP. Utilisez une nouvelle session pour traiter chaque requête. Utilisez `Session.merge()` ou `Session.saveOrUpdate()` pour synchroniser les objets avec la base de données.

Dans une architecture à deux couches, pensez à utiliser les contextes de persistance longue :

Les transactions de bases de données doivent être aussi courtes que possible pour une meilleure extensibilité. Cependant, il est souvent nécessaire d'implémenter de longues *transactions applicatives*, une simple unité de travail du point de vue de l'utilisateur. Une transaction applicative peut s'étaler sur plusieurs cycles de requêtes/réponses du client. Il est commun d'utiliser des objets détachés pour implémenter des transactions applicatives. Une alternative, extrêmement appropriée dans une architecture à deux couches, est de maintenir un seul contact de persistance ouvert (session) pour toute la durée de vie de la transaction applicative et simplement se déconnecter de la connexion JDBC à la fin de chaque requête, et se reconnecter au début de la requête suivante. Ne partagez jamais une seule session avec plus d'une transaction applicative, ou bien vous travaillerez avec des données périmées.

Considérez que les exceptions ne sont pas rattrapables :

Il s'agit plus d'une pratique obligatoire que d'une "meilleure pratique". Quand une exception intervient, il faut faire un rollback de la `Transaction` et fermer la `Session`. Sinon, Hibernate ne peut garantir l'intégrité des états persistants en mémoire. En particulier, n'utilisez pas `Session.load()` pour déterminer si une instance avec l'identifiant donné existe en base de données, à la place utilisez `Session.get()` ou une requête.

Préférez le chargement différé des associations :

Utilisez le chargement complet avec modération. Utilisez les proxies et les collections chargées tardivement pour la plupart des associations vers des classes qui ne sont pas susceptibles d'être complètement retenues dans le cache de second niveau. Pour les associations de classes en cache, où il y a une forte probabilité que l'élément soit en cache, désactivez explicitement le chargement par jointures ouvertes en utilisant `outer-`

`join="false"`. Lorsqu'un chargement par jointure ouverte est approprié pour un cas d'utilisation particulier, utilisez une requête avec un `left join fetch`.

Utilisez le pattern *d'une ouverture de session dans une vue*, ou une *phase d'assemblage* disciplinée pour éviter des problèmes avec des données non rapatriées :

Hibernate libère les développeurs de l'écriture fastidieuse des *objets de transfert de données* (DTO). Dans une architecture EJB traditionnelle, les DTO ont deux buts : premièrement, ils contournent le problème des beans entités qui ne sont pas sérialisables ; deuxièmement, ils définissent implicitement une phase d'assemblage où toutes les données utilisées par la vue sont rapatriées et organisées dans les DTO avant de retourner sous le contrôle de la couche de présentation. Hibernate élimine le premier but. Cependant, vous aurez encore besoin d'une phase d'assemblage (pensez à vos méthodes métier comme ayant un contrat strict avec la couche de présentation, en ce qui concerne les données disponibles dans les objets détachés) à moins que vous soyez préparés à garder le contexte de persistance (la session) ouvert à travers tout le processus de rendu de la vue. Ceci ne représente pas une limitation de Hibernate ! Au contraire c'est une exigence fondamentale d'un accès sécurisé aux données transactionnelles.

Pensez à abstraire votre logique métier d'Hibernate :

Cachez le mécanisme d'accès aux données (Hibernate) derrière une interface. Combinez les modèles *DAO* et *Thread Local Session*. Vous pouvez même avoir quelques classes persistées par du JDBC pur, associées à Hibernate via un `UserType` (ce conseil est valable pour des applications de taille respectables ; il n'est pas valable pour une application avec cinq tables).

N'utilisez pas d'associations de mapping exotiques :

Les utilisations appropriées de vraies associations plusieurs-à-plusieurs sont rares. La plupart du temps vous avez besoin d'informations additionnelles stockées dans la table d'association. Dans ce cas, il est préférable d'utiliser deux associations un-à-plusieurs vers une classe de liaisons intermédiaire. En fait, nous pensons que la plupart des associations sont de type un-à-plusieurs ou plusieurs-à-un, vous devez être très prudent lorsque vous utilisez toute autre association et vous demander si c'est vraiment nécessaire.

Préférez les associations bidirectionnelles :

Les associations unidirectionnelles sont plus difficiles à questionner. Dans une grande application, la plupart des associations devraient être navigables dans les deux directions dans les requêtes.

Considérations de portabilité des bases de données

26.1. Aspects fondamentaux de la portabilité

La portabilité des bases de données est un des atouts qui sont mis en avant pour vendre Hibernate (et plus largement le mappage objet/relationnel dans son ensemble). Il pourrait s'agir d'un utilisateur IT interne qui migre d'une base de données de fournisseur vers une autre, ou il pourrait s'agir d'un framework ou d'une application déployable consommant Hibernate pour cibler simultanément plusieurs produits de base de données par leurs utilisateurs. Quel que soit le scénario exact, l'idée de base est que vous souhaitez que HIBERNATE vous permette d'exécuter avec un certain nombre de bases de données sans modifications à votre code et idéalement sans modifications des métadonnées de mappage.

26.2. Dialecte

La première ligne de la portabilité d'Hibernate est le dialecte, qui est une spécialisation du contrat `org.hibernate.dialect.dialect`. Un dialecte encapsule toutes les différences selon lesquelles Hibernate doit communiquer avec une base de données particulière pour accomplir certaines tâches comme l'obtention d'une valeur de la séquence ou de structuration d'une requête SELECT. Hibernate regroupe un large éventail de dialectes pour la plupart des bases de données les plus communes. Si vous trouvez que votre base de données particulière n'en fait pas partie, il n'est pas difficile d'écrire votre propre dialecte.

26.3. Résolution de dialecte

À l'origine, Hibernate exigeait toujours que les utilisateurs spécifient quel dialecte utiliser. Dans le cas des utilisateurs qui cherchent à cibler simultanément plusieurs bases de données avec leur version, c'était problématique. Généralement cela amenait leurs utilisateurs à configurer le dialecte Hibernate ou à définir leur propre méthode de définition de cette valeur.

A partir de la version 3.2, Hibernate a introduit la détection automatiquement du dialecte à utiliser basé sur les `Java.SQL.DatabaseMetaData` obtenues à partir d'un `Java.SQL.Connexion` vers cette base de données. C'était beaucoup mieux, sauf que cette résolution a été limitée aux bases de données déjà connues d'Hibernate et elle n'était ni configurable, ni remplaçable.

Starting with version 3.3, Hibernate has a far more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the `org.hibernate.dialect.resolver.DialectResolver` which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

. The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies `org.hibernate.exception.JDBCConnectionException` as possibly being thrown. A `JDBCConnectionException` here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

Le bon côté de ces outils de résolution, c'est que les utilisateurs peuvent également enregistrer leurs propres outils de résolution personnalisés, qui seront traités avant les résolveurs Hibernate intégrés. Cette option peut être utile dans un certain nombre de situations différentes : elle permet une intégration aisée pour la détection automatique des dialectes au-delà de ceux qui sont livrés avec Hibernate lui-même ; elle vous permet de spécifier d'utiliser un dialecte personnalisé lorsqu'une base de données particulière est reconnue ; etc.. Pour enregistrer un ou plusieurs outils de résolution, il vous suffit de les spécifier (séparés par des virgules, des onglets ou des espaces) à l'aide du paramètre de configuration 'hibernate.dialect_resolvers' (voir la constante `DIALECT_RESOLVERS` sur `cfg.Environment org.Hibernate.`).

26.4. Générer les identifiants

When considering portability between databases, another important decision is selecting the identifier generation strategy you want to use. Originally Hibernate provided the *native* generator for this purpose, which was intended to select between a *sequence*, *identity*, or *table* strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targetting some databases which support *identity* generation and some which do not. *identity* generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value; it is what is known as a post-insert generation strategy because the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context it must then issue the insert immediately when the users requests the entity be associated with the session (like via `save()` e.g.) regardless of current transactional semantics.



Note

Hibernate was changed slightly once the implication of this was better understood so that the insert is delayed in cases where that is feasible.

The underlying issue is that the actual semantics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of *enhanced* [<http://in.relation.to/2082.lace>] identifier generators targetting portability in a much different way.



Note

There are specifically 2 bundled *enhanced* generators:

- `org.hibernate.id.enhanced.SequenceStyleGenerator`
- `org.hibernate.id.enhanced.TableGenerator`

The idea behind these generators is to port the actual semantics of the identifier value generation to the different databases. For example, the `org.hibernate.id.enhanced.SequenceStyleGenerator` mimics the behavior of a sequence on databases which do not support sequences by using a table.

26.5. Fonctions de base de données



Avertissement

This is an area in Hibernate in need of improvement. In terms of portability concerns, this function handling currently works pretty well from HQL; however, it is quite lacking in all other aspects.

SQL functions can be referenced in many ways by users. However, not all databases support the same set of functions. Hibernate, provides a means of mapping a *logical* function name to a delegate which knows how to render that particular function, perhaps even using a totally different physical function call.



Important

Techniquement, cet enregistrement de la fonction est g  r   par le biais de la classe `hibernate.dialect.function.SQLFunctionRegistry` org. qui est destin  e    permettre aux utilisateurs de fournir des d  finitions de fonction personnalis  e sans avoir    fournir un dialecte personnalis  . Ce comportement sp  cifique n'est pas encore enti  rement termin  .

Il est mis en oeuvre de telle sorte que les utilisateurs peuvent enregistrer des fonctions par programmation avec `org.Hibernate.cfg.Configuration` et ces fonctions seront reconnues pour HQL.

26.6. Type mappings

This section scheduled for completion at a later date...

References

- [PoEAA] *Patterns of Enterprise Application Architecture*. 0-321-12742-0. par Martin Fowler.
Copyright © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.
- [JPwH] *Java Persistence with Hibernate*. Second Edition of Hibernate in Action. 1-932394-88-5.
<http://www.manning.com/bauer2> . par Christian Bauer et Gavin King. Copyright © 2007
Manning Publications Co.. Manning Publications Co..

