Hibernate Annotations

Guide de référence

Version: 3.2.0.GA

Table des matières

Préface

Traducteur(s): Vincent Ricard

Hibernate, comme tous les autres outils de mapping objet/relationnel, nécessite des méta-données qui régissent la transformation des données d'une représentation vers l'autre (et vice versa). Dans Hibernate 2.x, les méta-données de mapping sont la plupart du temps déclarées dans des fichiers XML. Une autre option est XDoclet, qui utilise les annotations du code source Javadoc et un préprocesseur au moment de la compilation. Le même genre d'annotation est maintenant disponible avec le JDK standard, quoique plus puissant et mieux pris en charge par les outils. IntelliJ IDEA et Eclipse, par exemple, prennent en charge la complétion automatique et la coloration syntaxique des annotations du JDK 5.0. Les annotations sont compilées en bytecode et lues au moment de l'exécution (dans le cas d'Hibernate, au démarrage) en utilisant la réflexion, donc pas besoin de fichiers XML externes.

La spécification EJB3 reconnaît l'intérêt et le succès du paradigme du mapping objet/relationnel transparent. La spécification EJB3 standardise les APIs de base et les méta-données requises par n'importe quel mécanisme de persistance objet/relationnel. *Hibernate EntityManager* implémente les interfaces de programmation et les règles de cycle de vie telles que définies par la spécification de persistance EJB3. Avec *Hibernate Annotations*, ce wrapper implémente une solution de persistance EJB3 complète (et autonome) au-dessus du noyau mature d'Hibernate. Vous pouvez utiliser soit les trois ensembles, soit les annotations sans le cycle de vie et les interfaces de programmations EJB3, ou même Hibernate tout seul, selon les besoins techniques et fonctionnels de votre projet. Vous pouvez à tout moment recourir aux APIs natives d'Hibernate ou même, si besoin est, à celles de JDBC et au SQL.

Cette version est basée sur la dernière version de la spécification EJB 3.0 / JPA (alias JSP-220) et prend en charge toutes les fonctionnalités de la spécification (dont certaines optionnelles). La plupart des fonctionnalités d'Hibernate et des extensions sont aussi disponibles à travers des annotations spécifiques à Hibernate. Bien que la couverture d'Hibernate en termes de fonctionnalités soit maintenant très grande, certaines sont encore manquantes. Le but ultime est de tout couvrir. Voir la section JIRA "road map" pour plus d'informations.

Si vous utilisiez une version précédente d'Hibernate Annotations, veuillez regarder `http://www.hibernate.org/371.html` pour un guide de migration.

Chapitre 1. Configurer un projet avec des annotations

1.1. Prérequis

- Téléchargez et installez la distribution Hibernate Annotations à partir du site web d'Hibernate.
- *Cette version requiert Hibernate 3.2.0.GA ou supérieur. N'utilisez pas cette version d'Hibernate Annotations avec une version plus ancienne d'Hibernate 3.x !*
- Cette version est connue pour fonctionner avec le noyau 3.2.0.CR5 et 3.2.0.GA d'Hibernate.
- Assurez-vous que vous avez le JDK 5.0 d'installé. Vous pouvez bien sûr continuer à utiliser XDoclet et avoir certains des avantages des méta-données basées sur les annotations avec des versions plus anciennes du JDK. Notez que ce document décrit seulement les annotations du JDK 5.0 et que vous devez vous référer à la documentation de XDoclet pour plus d'informations.

1.2. Configuration

Tout d'abord, paramétrez votre classpath (après avoir créer un nouveau projet dans votre IDE favori) :

- Copiez toutes les bibliothèques du noyau Hibernate3 et toutes les bibliothèques tierces requises (voir lib/README.txt dans Hibernate).
- Copiez aussi `hibernate-annotations.jar` et `lib/ejb3-persistence.jar` de la distribution Hibernate Annotations dans votre classpath.
- Pour utiliser Chapitre 5, *Intégration de Lucene avec Hibernate*, ajouter le fichier jar de lucene.

Nous recommandons aussi un petit wrapper pour démarrer Hibernate dans un bloc statique d'initialisation, connu en tant que `HibernateUtil`. Vous pourriez avoir vu cette classe sous diverses formes dans d'autres parties de la documentation Hibernate. Pour prendre en charge Annotation vous devez modifier cette classe d'aide de la manière suivante :

```
package hello;

import org.hibernate.*;
import org.hibernate.cfg.*;
import test.*;
import test.animals.Dog;

public class HibernateUtil {

private static final SessionFactory sessionFactory;

    static {
        try {

            sessionFactory = new AnnotationConfiguration().buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }
```

```
    public static Session getSession()
            throws HibernateException {
        return sessionFactory.openSession();
    }
}
```

La partie intéressante ici est l'utilisation de `AnnotationConfiguration`. Les packages et les classes annotées sont déclarés dans votre fichier de configuration XML habituel (généralement `hibernate.cfg.xml`). Voici un équivalent de la déclaration ci-dessus :

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

        <hibernate-configuration>
          <session-factory>
            <mapping package="test.animals"/>
            <mapping class="test.Flight"/>
            <mapping class="test.Sky"/>
            <mapping class="test.Person"/>
            <mapping class="test.animals.Dog"/>
            <mapping resource="test/animals/orm.xml"/>
          </session-factory>
        </hibernate-configuration>
```

Notez que vous pouvez mélanger l'utilisation du fichier hbm.xml et celle des annotations. L'élément de ressource peut être un fichier hbm ou un descripteur de déploiement XML EJB3. Cette distinction est transparente pour votre procédure de configuration.

Alternativement, vous pouvez définir les classes annotées et des packages en utilisant l'API :

```
            sessionFactory = new AnnotationConfiguration()
                    .addPackage("test.animals") // le nom complet du package
                    .addAnnotatedClass(Flight.class)
                    .addAnnotatedClass(Sky.class)
                    .addAnnotatedClass(Person.class)
                    .addAnnotatedClass(Dog.class)
                    .addResource("test/animals/orm.xml")
                    .buildSessionFactory();
```

Vous pouvez aussi utiliser Hibernate EntityManager qui a son propre mécanisme de configuration. Veullez vous référer à la documentation de ce projet pour plus de détails.

Il n'y a pas d'autres différences dans la façon d'utiliser les APIs d'Hibernate, excepté ce changement de routine de démarrage ou le fichier de configuration. Vous pouvez utiliser votre méthode de configuration favorite pour d'autres propriétés (`hibernate.properties`, `hibernate.cfg.xml`, utilisation des APIs, etc). Vous pouvez même mélanger les classes persistantes annotées et des déclarations `hbm.cfg.xml` classiques avec la même `SessionFactory`. Vous ne pouvez cependant pas déclarer une classe plusieurs fois (soit avec les annotations, soit avec un fichier hbm.xml). Vous ne pouvez pas non plus mélanger des stratégies de configuration (hbm vs annotations) dans une hiérarchie d'entités mappées.

Pour faciliter la procédure de migration de fichiers hbm vers les annotations, le mécanisme de configuration détecte la duplication de mappings entre les annotations et les fichiers hbm. Les classes décrites dans les fichiers hbm se voient alors affecter une priorité plus grande que les classes annotées. Vous pouvez changer cette priorité avec la propriété `hibernate.mapping.precedence`. La valeur par défaut est : `hbm, class` ; la changer en : `class, hbm` donne alors la priorité aux classes annotées lorsqu'un conflit survient.

Chapitre 2. Entity Beans

2.1. Introduction

Cette section couvre les annotations entity bean EJB 3.0 (alias JPA) et les extensions spécifiques à Hibernate.

2.2. Mapping avec les annotations EJB3/JPA

Les entités EJB3 sont des POJOs ordinaires. En fait, ils représentent exactement le même concept que les entités de persistance Hibernate. Leur mapping est défini à travers les annotations du JDK 5.0 (une syntaxe de descripteur XML pour la surcharge est définie dans la spécification EJB3). Les annotations peuvent être divisées en deux catégories, les annotations de mapping logique (vous permettant de décrire le modèle objet, les associations de classe, etc) et les annotations de mapping physique (décrivant le schéma physique, les tables, les colonnes, les index, etc). Nous mélangerons les annotations des deux catégories dans les exemples de code.

Les annotations EJB3 sont dans le package `javax.persistence.*`. La plupart des IDE compatibles JDK 5 (comme Eclipse, IntelliJ IDEA et Netbeans) peuvent auto-compléter les interfaces et les attributes d'annotation pour vous (même sans module "EJB3" spécifique, puisque les annotations EJB3 sont des annotations ordinaires de JDK 5).

Pour plus d'exemples concrets, lisez le tutorial EJB 3.0 de JBoss ou parcourez la suite de tests d'Hibernate Annotations. La plupart des tests unitaires ont été conçus pour représenter un exemple concret et être une source d'inspiration.

2.2.1. Déclarer un entity bean

Chaque classe POJO persistante liée est un entity bean et est déclarée en utilisant l'annotation `@Entity` (au niveau de la classe) :

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

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

`@Entity` déclare la classe comme un entity bean (ie une classe POJO persistante), `@Id` déclare la propriété identifiante de cet entity bean. Les autres déclarations de mapping sont implicites. Ce concept de déclaration par exception est un composant essentiel de la nouvelle spécification EJB3 et une amélioration majeure. La classe Flight est mappée sur la table Flight, en utilisant la colonne id comme colonne de la clef primaire.

Selon que vous annotez des champs ou des méthodes, le type d'accès utilisé par Hibernate sera `field` ou `property`. La spécification EJB3 exige que vous déclariez les annotations sur le type d'élément qui sera accédé, c'est-à-dire le getter si vous utilisez l'accès `property`, le champ si vous utilisez l'accès `field`. Mélanger des EJB3 annotations dans les champs et les méthodes devrait être évité. Hibernate devinera le type d'accès de l'identifiant à partir de la position d'`@Id` ou d'`@EmbeddedId`.

2.2.1.1. Définir la table

`@Table` est positionnée au niveau de la classe ; cela vous permet de définir le nom de la table, du catalogue et du schéma pour le mapping de votre entity bean. Si aucune `@Table` n'est définie les valeurs par défaut sont utilisées : le nom de la classe de l'entité (sans le nom de package).

```
@Entity
@Table(name="tbl_sky")
public class Sky implements Serializable {
...
```

L'élément `@Table` contient aussi un attribut `schema` et un attribut `catalog`, si vous avez besoin de les définir. Vous pouvez aussi définir des contraintes d'unicité sur la table en utilisant l'annotation `@UniqueConstraint` en conjonction avec `@Table` (pour une contrainte d'unicité n'impliquant qu'une seule colonne, référez-vous à `@Column`).

```
@Table(name="tbl_sky",
    uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})}
)
```

Une contrainte d'unicité est appliquée au tuple {month, day}. Notez que le tableau `columnNames` fait référence aux noms logiques des colonnes.
Le nom logique d'une colonne est défini par l'implémentation de NamingStrategy d'Hibernate. La stratégie de nommage EJB3 par défaut utilise le nom de colonne physique comme nom de colonne logique. Notez qu'il peut être différent du nom de la propriété (si le nom de colonne est explicite). A moins que vous surchargiez la stratégie de nommage, vous ne devriez pas vous soucier de ça.

2.2.1.2. Versionner pour un contrôle de concurrence optimiste

Vous pouvez ajouter un contrôle de concurrence optimiste à un entity bean en utilisant l'annotation `@Version` :

```
@Entity
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

La propriété de version sera mappée sur la colonne `OPTLOCK`, et le gestionnaire d'entités l'utilisera pour détecter des conflits lors des mises à jour (prévenant des pertes de données lors de mises à jours que vous pourriez voir avec la stratégie du last-commit-wins).

La colonne de version peut être un numérique (solution recommandée) ou un timestamp comme pour la spécification EJB3. Hibernate prend en charge n'importe quel type fourni que vous définissez et implémentez avec la classe `UserVersionType` appropriée.

2.2.2. Mapping de simples propriétés

2.2.2.1. Déclarer des mappings de propriétés élémentaires

Chaque propriété (champ ou méthode) non statique non transient d'un entity bean est considérée persistante, à

moins que vous l'annotiez comme `@Transient`. Ne pas avoir d'annotation pour votre propriété est équivalent à l'annotation `@Basic`. L'annotation `@Basic` vous permet de déclarer la stratégie de récupération pour une propriété :

```
public transient int counter; // propriété transient

private String firstname; // propriété persistante

@Transient
String getLengthInMeter() { ... } // propriété transient

String getName() {... } // propriété persistante

@Basic
int getLength() { ... } // propriété persistante

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // propriété persistante

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // propriété persistante

@Enumerated(STRING)
Starred getNote() { ... } // enum persistée en tant que String dans la base de données
```

`counter`, un champ transient, et `lengthInMeter`, une méthode annotée comme `@Transient`, seront ignorés par le gestionnaire d'entités. Les propriétés `name`, `length`, et `firstname` sont mappées comme persistantes et à charger immédiatement (ce sont les valeurs par défaut pour les propriétés simples). La valeur de la propriété `detailedComment` sera chargée à partir de la base de données dès que la propriété de l'entité sera accédée pour la première fois. En général vous n'avez pas besoin de marquer de simples propriétés comme "à charger à la demande" (NdT: lazy) (à ne pas confondre avec la récupération d'association "lazy").

Note

Pour activer la récupération à la demande au niveau de la propriété, vos classes doivent être instrumentées : du bytecode est ajouté au code original pour activer cette fonctionnalité, veuillez vous référer à la documentation de référence d'Hibernate. Si vos classes ne sont pas instrumentées, le chargement à la demande au niveau de la propriété est silencieusement ignoré.

L'alternative recommandée est d'utiliser la capacité de projection de JPA-QL ou des requêtes Criteria.

EJB3 prend en charge le mapping de propriété de tous les types élémentaires pris en charge par Hibernate (tous les types de base Java, leur wrapper respectif et les classes sérialisables). Hibernate Annotations prend en charge le mapping des types Enum soit vers une colonne ordinale (en stockant le numéro ordinal de l'enum), soit vers une colonne de type chaîne de caractères (en stockant la chaîne de caractères représentant l'enum) : la représentation de la persistance, par défaut ordinale, peut être surchargée grâce à l'annotation `@Enumerated` comme montré avec la propriété `note` de l'exemple.

Dans les APIs core de Java, la précision temporelle n'est pas définie. Lors du traitement de données temporelles vous pourriez vouloir décrire la précision attendue dans la base de données. Les données temporelles peuvent avoir une précision de type `DATE`, `TIME`, ou `TIMESTAMP` (c'est-à-dire seulement la date, seulement l'heure, ou les deux). Utilisez l'annotation `@Temporal` pour ajuster cela.

`@Lob` indique que la propriété devrait être persistée dans un Blob ou un Clob selon son type : `java.sql.Clob`, `Character[]`, `char[]` et java.lang.`String` seront persistés dans un Clob. `java.sql.Blob`, `Byte[]`, `byte[]` et les types sérialisables seront persistés dans un Blob.

```
@Lob
```

```
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
}
```

Si le type de la propriété implémente `java.io.Serializable` et n'est pas un type de base, et si la propriété n'est pas annotée avec `@Lob`, alors le type Hibernate `serializable` est utilisé.

2.2.2.2. Déclarer des attributs de colonne

La(les) colonne(s) utilisée(s) pour mapper une propriété peuvent être définies en utilisant l'annotation `@Column`. Utilisez-la pour surcharger les valeurs par défaut (voir la spécification EJB3 pour plus d'informations sur les valeurs par défaut). Vous pouvez utilisez cette annotation au niveau de la propriété pour celles qui sont :

- pas du tout annotées
- annotées avec `@Basic`
- annotées avec `@Version`
- annotées avec `@Lob`
- annotées avec `@Temporal`
- annotées avec `@org.hibernate.annotations.CollectionOfElements` (pour Hibernate uniquement)

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

La propriété `name` est mappée sur la colonne `flight_name`, laquelle ne peut pas avoir de valeur nulle, a une longueur de 50 et ne peut pas être mise à jour (rendant la propriété immuable).

Cette annotation peut être appliquée aux propriétés habituelles ainsi qu'aux propriétés `@Id` ou `@Version`.

```
@Column(
    name="columnName";                                (1)
    boolean unique() default false;                   (2)
    boolean nullable() default true;                  (3)
    boolean insertable() default true;                (4)
    boolean updatable() default true;                 (5)
    String columnDefinition() default "";             (6)
    String table() default "";                        (7)
    int length() default 255;                         (8)
    int precision() default 0; // decimal precision   (9)
    int scale() default 0; // decimal scale
```

(1) `name` (optionnel) : le nom de la colonne (par défaut le nom de la propriété)
(2) `unique` (optionnel) : indique si la colonne fait partie d'une contrainte d'unicité ou non (par défaut false)
(3) `nullable` (optionnel) : indique si la colonne peut avoir une valeur nulle (par défaut false).

(4) `insertable` (optionnel) : indique si la colonne fera partie de la commande insert (par défaut true)
(5) `updatable` (optionnel) : indique si la colonne fera partie de la commande update (par défaut true)
(6) `columnDefinition` (optionnel) : surcharge le fragment DDL sql pour cette colonne en particulier (non portable)
(7) `table` (optionnel) : définit la table cible (par défaut la table principale)
(8) `length` (optionnel) : longueur de la colonne (par défaut 255)
(8) `precision` (optionnel) : précision décimale de la colonne (par défaut 0)
(10) `scale` (optionnel) : échelle décimale de la colonne si nécessaire (par défaut 0)

2.2.2.3. Objets embarqués (alias composants)

Il est possible de déclarer un composant embarqué à l'intérieur d'une entité et même de surcharger le mapping de ses colonnes. Les classes de composant doivent être annotées au niveau de la classe avec l'annotation `@Embeddable`. Il est possible de surcharger le mapping de colonne d'un objet embarqué pour une entité particulière en utilisant les annotations `@Embedded` et `@AttributeOverride` sur la propriété associée :

```
@Entity
public class Person implements Serializable {

    // Composant persistant utilisant les valeurs par défaut
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    String city;
    Country nationality; // par de surcharge ici
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }


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

Un objet embarquable hérite du type d'accès de son entité d'appartenance (notez que vous pouvez surcharger cela en utilisant les annotations spécifiques à Hibernate `@AccessType`, voir Extensions d'Hibernate Annotation).

L'entity bean `Person` a deux propriétés composant, `homeAddress` et `bornIn`. La propriété `homeAddress` n'a pas été annotée, mais Hibernate devinera que c'est un composant persistant en cherchant l'annotation `@Embeddable`

dans la classe Address. Nous surchargeons aussi le mapping d'un nom de colonne (pour `bornCountryName`) avec les annotations `@Embedded` et `@AttributeOverride` pour chaque attribut mappé de `Country`. Comme vous pouvez le voir, `Country` est aussi un composant imbriqué de `Address`, utilisant de nouveau la détection automatique d'Hibernate et les valeurs par défaut EJB3. Surcharger des colonnes d'objets embarqués d'objets (eux-mêmes) embarqués n'est actuellement pas pris en charge par la spécification EJB3, cependant, Hibernate Annotations le prend en charge à travers des expressions séparées par des points.

```
@Embedded
@AttributeOverrides( {
        @AttributeOverride(name="city", column = @Column(name="fld_city") )
        @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
        @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
        // les colonnes de nationality dans homeAddress sont surchargées
} )
Address homeAddress;
```

Hibernate Annotations prend en charge une fonctionnalité de plus qui n'est pas explicitement prise en charge par la spécification EJB3. Vous pouvez annoter un objet embarqué avec l'annotation `@MappedSuperclass` pour rendre les propriétés de la classe parente persistantes (voir `@MappedSuperclass` pour plus d'informations).

Alors que ce n'est pas pris en charge par la spécification EJB3, Hibernate Annotations vous permet d'utiliser les annotations d'association dans un objet embarquable (ie `@*ToOne` ou `@*ToMany`). Pour surcharger les colonnes de l'association vous pouvez utiliser `@AssociationOverride`.

Si vous voulez avoir le même type d'objet embarquable deux fois dans la même entité, le nom de colonne par défaut ne fonctionnera pas : au moins une des colonnes devra être explicitée. Hibernate va au-delà de la spécification EJB3 et vous permet d'améliorer le mécanisme par défaut avec `NamingStrategy`. `DefaultComponentSafeNamingStrategy` est une petite amélioration par rapport à la stratégie par défaut `EJB3NamingStrategy` qui permet aux objets embarqués de fonctionner avec leur valeur par défaut même s'ils sont utilisés deux fois dans la même entité.

2.2.2.4. Valeurs par défaut des propriétés non annotées

Si une propriété n'est pas annotée, les règles suivantes s'appliquent :

- Si la propriété est de type simple, elle est mappée comme @Basic
- Sinon, si le type de la propriété est annoté comme @Embeddable, elle est mappée comme @Embedded
- Sinon, si le type de la propriété est Serializable, elle est mappée comme @Basic vers une colonne contenant l'objet sous sa forme sérialisée
- Sinon, si le type de la propriété est java.sql.Clob ou java.sql.Blob, elle est mappée comme @Lob avec le LobType approprié

2.2.3. Mapper des propriétés identifiantes

L'annotation `@Id` vous permet de définir quelle propriété identifie votre entity bean. Cette propriété peut être positionnée par l'application elle-même ou générée par Hibernate (préféré). Vous pouvez définir la stratégie de génération de l'identifiant grâce à l'annotation `@GeneratedValue` :

- AUTO - soit la colonne identity, soit la séquence, soit la table selon la base de données sous-jacente
- TABLE - table contenant l'id

- IDENTITY - colonne identity
- SEQUENCE - séquence

Hibernate fournit plus de générateurs d'identifiant que les simples générateurs EJB3. Vérifiez Extensions d'Hibernate Annotation pour plus d'informations.

L'exemple suivant montre un générateur par séquence utilisant la configuration SEQ_STORE (voir plus bas) :

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
public Integer getId() { ... }
```

L'exemple suivant utilise le générateur identity :

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
public Long getId() { ... }
```

Le générateur `AUTO` est le type préféré pour les applications portables (vers différentes base de données). La configuration de la génération d'identifiant peut être partagée par différents mappings `@Id` avec l'attribut du générateur. Il y a différentes configurations disponibles avec `@SequenceGenerator` et `@TableGenerator`. La portée d'un générateur peut être l'application ou la classe. Les générateurs définis dans les classes ne sont pas visibles à l'extérieur de la classe et peuvent surcharger les générateurs de niveau applicatif. Les générateurs de niveau applicatif sont définis au niveau XML (voir Chapitre 3, *Surcharger des méta-données à travers du XML*) :

```
<table-generator name="EMP_GEN"
            table="GENERATOR_TABLE"
            pk-column-name="key"
            value-column-name="hi"
            pk-column-value="EMP"
            allocation-size="20"/>

// et l'annotation équivalente

@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)

<sequence-generator name="SEQ_GEN"
    sequence-name="my_sequence"
    allocation-size="20"/>

// et l'annotation équivalente

@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
```

Si JPA XML (comme `META-INF/orm.xml`) est utilisé pour définir les générateurs, `EMP_GEN` et `SEQ_GEN` sont des générateurs de niveau applicatif. `EMP_GEN` définit un générateur d'identifiant basé sur une table utilisant

l'algorithme hilo avec un `max_lo` de 20. La valeur haute est conservée dans une `table` "`GENERATOR_TABLE`". L'information est gardée dans une ligne où la colonne `pkColumnName` ("clef") est égale à `pkColumnValue` "`EMP`" et une colonne `valueColumnName` "`hi`" contient la prochaine valeur haute utilisée.

`SEQ_GEN` définit un générateur par séquence utilisant une séquence nommée `my_sequence`. La taille d'allocation utilisée pour cet algorithme hilo basé sur une séquence est 20. Notez que cette version d'Hibernate Annotations ne gère pas `initialValue` dans le générateur par séquence. La taille par défaut de l'allocation est 50, donc si vous voulez utiliser une séquence et récupérer la valeur chaque fois, vous devez positionner la taille de l'allocation à 1.

Note

La définition au niveau package n'est plus prise en charge par la spécification EJB 3.0. Vous pouvez cependant utiliser `@GenericGenerator` au niveau du package (voir Section 2.4.2, « Identifiant »).

Le prochain exemple montre la définition d'un générateur par séquence dans la portée d'une classe :

```
@Entity
@javax.persistence.SequenceGenerator(
    name="SEQ_STORE",
    sequenceName="my_sequence"
)
public class Store implements Serializable {
    private Long id;

    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
    public Long getId() { return id; }
}
```

Cette classe utilisera une séquence nommée my_sequence et le générateur SEQ_STORE n'est pas visible dans les autres classes. Notez que vous pouvez regarder les tests unitaires d'Hibernate Annotations dans le package org.hibernate.test.metadata.id pour plus d'exemples.

Vous pouvez définir une clef primaire composée à travers différentes syntaxes :

- annote la propriété du composant comme @Id et rend la classe du composant @Embeddable
- annote la propriété du composant comme @EmbeddedId
- annote la classe comme @IdClass et annote chaque propriété de l'entité impliquée dans la clef primaire avec @Id

Bien qu'assez commun pour le développeur EJB2, `@IdClass` est probablement nouveau pour les utilisateurs d'Hibernate. La classe de la clef primaire composée correspond aux multiples champs ou propriétés de l'entité ; de plus, les noms des champs ou propriétés de la clef primaire et ceux de l'entité doivent correspondre ; et enfin, leur type doit être le même. Regardons un exemple :

```
@Entity
@IdClass(FootballerPk.class)
public class Footballer {
    // partie de la clef
    @Id public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
```

```
    // partie de la clef
    @Id public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    public String getClub() {
        return club;
    }

    public void setClub(String club) {
        this.club = club;
    }

    // implémentation appropriée de equals() et hashCode()
}

@Embeddable
public class FootballerPk implements Serializable {
    // même nom et même type que dans Footballer
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    // même nom et même type que dans Footballer
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    // implémentation appropriée de equals() et hashCode()
}
```

Comme vous pouvez le voir, `@IdClass` pointe vers la classe de la clef primaire correspondante.

Bien que ce ne soit pas pris en charge par la spécification EJB3, Hibernate vous permet de définir des associations à l'intérieur d'un identifiant composé. Pour cela, utilisez simplement les annotations habituelles.

```
@Entity
@AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan_id") )
public class TvMagazin {
    @EmbeddedId public TvMagazinPk id;
    @Temporal(TemporalType.TIME) Date time;
}

@Embeddable
public class TvMagazinPk implements Serializable {
    @ManyToOne
    public Channel channel;
    public String name;
    @ManyToOne
    public Presenter presenter;
}
```

2.2.4. Mapper l'héritage

EJB3 prend en charge les trois types d'héritage :

- Stratégie d'une table par classe concrète : l'élément <union-class> dans Hibernate
- Stratégie d'une seule table par hiérarchie de classe : l'élément <subclass> dans Hibernate
- Stratégie d'une table par classe fille : l'élément <joined-subclass> dans Hibernate

La stratégie choisie est déclarée au niveau de la classe de l'entité la plus haute dans la hiérarhie en utilisant l'annotation `@Inheritance`.

> **Note**
>
> Annoter des interfaces n'est pour le moment pas pris en charge.

2.2.4.1. Une table par classe concrète

Cette stratégie a beaucoup d'inconvénients (surtout avec les requêtes polymorphiques et les associations) expliqués dans la spécification EJB3, la documentation de référence d'Hibernate, Hibernate in Action, et plusieurs autres endroits. Hibernate en contourne la plupart en implémentant cette stratégie en utilisant des requêtes `SQL UNION`. Elle est habituellement utilisée pour le niveau le plus haut d'une hiérarchie d'héritage :

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable {
```

Cette stratégie prend en charge les associations de un vers plusieurs bidirectionnelles. Cette stratégie ne prend pas en charge la stratégie de générateur `IDENTITY` : l'identifiant doit être partagé par plusieurs tables. Par conséquent, lors de l'utilisation de cette stratégie, vous ne devriez pas utilisez `AUTO` ni `IDENTITY`.

2.2.4.2. Une seule table par hiérarchie de classe

Toutes les propriétés de toutes les classes parentes et classes filles sont mappées dans la même table, les instances sont différenciées par une colonne spéciale discriminante :

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

`Plane` est la classe parente, elle définit la stratégie d'héritage `InheritanceType.SINGLE_TABLE`. Elle définit aussi la colonne discriminante avec l'annotation `@DiscriminatorColumn`, une colonne discriminante peut aussi définir le type du discriminant. Finalement, l'annotation `@DiscriminatorValue` définit la valeur utilisée pour différencier une classe dans la hiérarchie. Tous ces attributs ont des valeurs par défaut sensées. Le nom par défaut de la colonne discriminante est `DTYPE`. La valeur discriminante par défaut est le nom de l'entité (comme

défini dans `@Entity.name`) avec le type `DiscriminatorType.STRING`. `A320` est une classe fille ; vous devez seulement définir la valeur discriminante si vous ne voulez pas utiliser la valeur par défaut. La stratégie et le type du discriminant sont implicites.

`@Inheritance` et `@DiscriminatorColumn` devraient seulement être définies sur l'entité la plus haute de la hiérarchie.

2.2.4.3. Une table par classe fille

Les annotations `@PrimaryKeyJoinColumn` et `@PrimaryKeyJoinColumns` définissent la (les) clef(s) primaire(s) de la table de la classe fille jointe :

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Boat implements Serializable { ... }

@Entity
public class Ferry extends Boat { ... }

@Entity
@PrimaryKeyJoinColumn(name="BOAT_ID")
public class AmericaCupClass  extends Boat { ... }
```

Toutes les entités ci-dessus utilisent la stratégie `JOINED`, la table `Ferry` est jointe avec la table `Boat` en utilisant les mêmes noms de clef primaire. La table `AmericaCupClass` est jointe avec `Boat` en utilisant la condition de jointure `Boat.id = AmericaCupClass.BOAT_ID`.

2.2.4.4. Héritage de propriétés des classes parentes

Il est parfois utile de partager des propriétés communes à travers une classe technique ou métier sans l'inclure comme une entité habituelle (c'est-à-dire aucune table spécifique pour cette entité). Pour cela, vous pouvez les mapper comme `@MappedSuperclass`.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

En base de données, cette hiérarchie sera représentée comme une table `Order` ayant les colonnes `id`, `lastUpdate` et `lastUpdater`. Les mappings de propriété de la classe parente embarquée sont copiés dans les classes filles de l'entité. Souvenez-vous que la classe parente embarquable n'est cependant pas la racine de la hiérarchie.

Note

Les propriétés des classes parentes non mappées comme `@MappedSuperclass` sont ignorées.

Note

Le type d'accès (champ ou méthode) est hérité de l'entité racine, à moins que vous utilisiez l'annotation Hibernate `@AccessType`.

Note

La même notion peut être appliquée aux objets `@Embeddable` pour persister des propriétés de leurs classes parentes. Vous avez aussi besoin d'utiliser `@MappedSuperclass` pour faire ça (cependant cela ne devrait pas être considéré comme une fonctionnalité EJB3 standard).

Note

Il est permis de marquer une classe comme `@MappedSuperclass` dans le milieu d'une hiérarchie d'héritage mappée.

Note

Toute classe de la hiérarchie non annotée avec `@MappedSuperclass` ou `@Entity` sera ignorée.

Vous pouvez surcharger des colonnes définies dans des entités parentes au niveau de l'entité racine en utilisant l'annotation `@AttributeOverride`.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride( name="propulsion", joinColumns = @JoinColumn(name="fld_propulsion_fk") )
public class Plane extends FlyingObject {
    ...
}
```

La propriété `altitude` sera persistée dans la colonne `fld_altitude` de la table `Plane` et l'association `propulsion` sera matérialisée dans la colonne de clef étrangère `fld_propulsion_fk`.

Vous pouvez définir `@AttributeOverride`(s) et `@AssociationOverride`(s) sur des classes `@Entity`, des classes `@MappedSuperclass` et des propriétés pointant vers un objet `@Embeddable`.

2.2.5. Mapper des associations/relations d'entity beans

2.2.5.1. One-to-one

Vous pouvez associer des entity beans avec une relation one-to-one en utilisant `@OneToOne`. Il y a trois cas pour les associations one-to-one : soit les entités associées partagent les mêmes valeurs de clef primaire, soit une clef

étrangère est détenue par une des entités (notez que cette colonne de clef étrangère dans la base de données devrait être avoir une contrainte d'unicité pour simuler la cardinalité one-to-one), soit une table d'association est utilisée pour stocker le lien entre les 2 entités (une contrainte d'unicité doit être définie sur chaque clef étrangère pour assurer la cardinalité un à un).

Tout d'abord, nous mappons une véritable association one-to-one en utilisant des clefs primaires partagées :

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    public Heart getHeart() {
        return heart;
    }
    ...
}
```

```
@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

L'association un à un est activée en utilisant l'annotation `@PrimaryKeyJoinColumn`.

Dans l'exemple suivant, les entités associées sont liées à travers une clef étrangère :

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

Un `Customer` est lié à un `Passport`, avec une colonne de clef étrangère nommée `passport_fk` dans la table `Customer`. La colonne de jointure est déclarée avec l'annotation `@JoinColumn` qui ressemble à l'annotation `@Column`. Elle a un paramètre de plus nommé `referencedColumnName`. Ce paramètre déclare la colonne dans l'entité cible qui sera utilisée pour la jointure. Notez que lors de l'utilisation de `referencedColumnName` vers une colonne qui ne fait pas partie de la clef primaire, la classe associée doit être `Serializable`. Notez aussi que `referencedColumnName` doit être mappé sur une propriété ayant une seule colonne lorsqu'elle pointe vers une colonne qui ne fait pas partie de la clef primaire (d'autres cas pourraient ne pas fonctionnner).

L'association peut être bidirectionnelle. Dans une relation bidirectionnelle, une des extrémités (et seulement une) doit être la propriétaire : la propriétaire est responsable de la mise à jour des colonnes de l'association. Pour déclarer une extrémité comme *non* responsable de la relation, l'attribut `mappedBy` est utilisé. `mappedBy` référence le nom de la propriété de l'association du côté du propriétaire. Dans notre cas, c'est `passport`. Comme

vous pouvez le voir, vous ne devez (absolument) pas déclarer la colonne de jointure puisqu'elle a déjà été déclarée du côté du propriétaire.

Si aucune `@JoinColumn` n'est déclarée du côté du propriétaire, les valeurs par défaut s'appliquent. Une(des) colonne(s) de jointure sera(ont) créée(s) dans la table propriétaire, et son(leur) nom sera la concaténation du nom de la relation du côté propriétaire, _ (underscore), et le nom de la (des) colonne(s) de la clef primaire du propriétaire. Dans cet exemple `passport_id` parce que le nom de la propriété est `passport` et la colonne identifiante de `Passport` est `id`.

La troisième possibilité (utilisant une table d'association) est très exotique.

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "CustomerPassports"
        joinColumns = @JoinColumn(name="customer_fk"),
        inverseJoinColumns = @JoinColumns(name="passport_fk")
    )
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

Un `Customer` est lié à un `Passport` à travers une table d'association nommée `CustomerPassports` ; cette table d'association a une colonne de clef étrangère nommée `passport_fk` pointant vers la table `Passport` (matérialisée par l'attribut `inverseJoinColumn`), et une colonne de clef étrangère nommée `customer_fk` pointant vers la table `Customer` (matérialisée par l'attribut `joinColumns`).

Vous devez déclarer le nom de la table de jointure et les colonnes de jointure explicitement dans un tel mapping.

2.2.5.2. Many-to-one

Les associations Many-to-one sont déclarées au niveau de la propriété avec l'annotation `@ManyToOne` :

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

L'attribut `@JoinColumn` est optionnel, la valeur par défaut est comme l'association un à un, la concaténation du nom de la relation du côté propriétaire, _ (underscore), et le nom de la colonne de la clef primaire du côté propriétaire. Dans cet exemple, `company_id` parce que le nom de la propriété est `company` et la colonne identifiante de Company est `id`.

`@ManyToOne` a un paramètre nommé `targetEntity` qui décrit le nom de l'entité cible. Généralement, vous ne

devriez pas avoir besoin de ce paramètre puisque la valeur par défaut (le type de la propriété qui stocke l'association) est correcte dans la plupart des cas. Il est cependant utile lorsque vous souhaitez retourner une interface plutôt qu'une entité normale.

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
    ...
```

Vous pouvez sinon mapper une association plusieurs à un avec une table d'association. Cette association décrite par l'annotation `@JoinTable` contiendra une clef étrangère référençant la table de l'entité (avec `@JoinTable.joinColumns`) et une clef étrangère référençant la table de l'entité cible (avec `@JoinTable.inverseJoinColumns`).

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumns(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

2.2.5.3. Collections

2.2.5.3.1. Vue d'ensemble

Vous pouvez mapper des `Collections`, des `Lists` (ie des listes ordonnées, pas des listes indexées), des `Maps` et des `Sets`. La spécification EJB3 décrit comment mapper une liste ordonnée (ie une liste ordonnée au chargement) en utilisant l'annotation `@javax.persistence.OrderBy` : pour ordonner la collection, cette annotation prend en paramètre une liste de propriétés (de l'entité cible) séparées par des virgules (p. ex. `firstname asc, age desc`) ; si la chaîne de caractères est vide, la collection sera ordonnée par les identifiants. Pour le moment `@OrderBy` fonctionne seulement sur des collections n'ayant pas de table d'association. Pour les véritables collections indexées, veuillez vous référer à Extensions d'Hibernate Annotation. EJB3 vous permet de mapper des `Maps` en utilisant comme clef une des propriétés de l'entité cible avec `@MapKey(name="myProperty")` (myProperty est un nom de propriété de l'entité cible). Lorsque vous utilisez `@MapKey` sans nom de propriété, la clef primaire de l'entité cible est utilisée. La clef de la map utilise la même colonne que celle pointée par la propriété : il n'y a pas de colonne supplémentaire définie pour la clef de la map, et c'est normal puisque la clef de la map représente en fait un propriété de la cible. Faites attention qu'une fois chargée, la clef n'est plus synchronisée avec la propriété, en d'autres mots, si vous modifiez la valeur de la propriété, la clef ne sera pas changée automatiquement dans votre modèle Java (pour une véritable prise en charge des maps veuillez vous référer à Extensions d'Hibernate Annotation). Beaucoup de gens confondent les capacités de `<map>` et celles de `@MapKey`. Ce sont deux fonctionnalités différentes. `@MapKey` a encore quelques limitations, veuillez vous référer au forum ou au système de suivi de bogues JIRA pour plus d'informations.

Hibernate a plusieurs notions de collections.

Tableau 2.1. Sémantique des collections

Sémantique	Représentation Java	Annotations
Sémantique de Bag	java.util.List, java.util.Collection	@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany
Sémantique de Bag avec une clef primaire (sans les limitations de la sémantique de Bag)	java.util.List, java.util.Collection	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et @CollectionId
Sémantique de List	java.util.List	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et @org.hibernate.annotations.IndexColumn
Sémantique de Set	java.util.Set	@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany
Sémantique de Map	java.util.Map	(@org.hibernate.annotations.CollectionOfElements ou @OneToMany ou @ManyToMany) et (rien ou @org.hibernate.annotations.MapKey/MapKeyManyToMany pour une véritable prise en charge des maps, ou @javax.persistence.MapKey

Donc spécifiquement, les collections java.util.List sans @org.hibernate.annotations.IndexColumn vont être considérées commes des bags.

Les collections de types primitifs, de types core ou d'objets embarqués ne sont pas prises en charge par la spécification EJB3. Cependant Hibernate Annotations les autorise (voir Extensions d'Hibernate Annotation).

```
@Entity public class City {
    @OneToMany(mappedBy="city")
    @OrderBy("streetName")
    public List<Street> getStreets() {
        return streets;
    }
...
}

@Entity public class Street {
    public String getStreetName() {
        return streetName;
    }

    @ManyToOne
    public City getCity() {
```

```
        return city;
    }
    ...
}


@Entity
public class Software {
    @OneToMany(mappedBy="software")
    @MapKey(name="codeName")
    public Map<String, Version> getVersions() {
        return versions;
    }
...
}

@Entity
@Table(name="tbl_version")
public class Version {
    public String getCodeName() {...}

    @ManyToOne
    public Software getSoftware() { ... }
...
}
```

Donc `City` a une collection de `Streets` qui sont ordonnées par `streetName` (de `Street`) lorsque la collection est chargée. `Software` a une map de `Versions` dont la clef est `codeName` de `Version`.

A moins que la collection soit une "generic", vous devrez définir `targetEntity`. C'est un attribut de l'annotation qui prend comme valeur la classe de l'entité cible.

2.2.5.3.2. One-to-many

Les associations one-to-many sont déclarées au niveau propriété avec l'annotation `@OneToMany`. Les associations un à plusieurs peuvent être bidirectionnelles.

2.2.5.3.2.1. Relation bidirectionnelle

Puisque les associations plusieurs à un sont (presque) toujours l'extrémité propriétaire de la relation bidirectionnelle dans la spécification EJB3, l'association un à plusieurs est annotée par `@OneToMany(mappedBy=...)`.

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` a une relation bidirectionnelle un à plusieurs avec `Soldier` à travers la propriété `troop`. Vous ne devez pas définir de mapping physique à l'extrémité de `mappedBy`.

Pour mapper une relation bidirectionnelle un à plusieurs, avec l'extrémité one-to-many comme extrémité propriétaire, vous devez enlever l'élément `mappedBy` et marquer l'annotation `@JoinColumn` de l'extrémité

plusieurs à un comme ne pouvant pas être insérée et ni mise à jour. Cette solution n'est certainement pas optimisée et produira quelques commandes UPDATE supplémentaires.

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") // nous avons besoin de dupliquer l'information physique
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

2.2.5.3.2.2. Relation unidirectionnelle

Une relation un à plusieurs unidirectionnelle utilisant une colonne de clef étrangère de l'entité propriétaire n'est pas si commune, réellement recommandée. Nous vous conseillons fortement d'utiliser une table de jointure pour cette sorte d'association (comme expliqué dans la prochaine section). Cette sorte d'association est décrite à travers `@JoinColumn`.

```
@Entity
public class Customer implements Serializable {
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    public Set<Ticket> getTickets() {
    ...
}

@Entity
public class Ticket implements Serializable {
    ... // pas de relation bidirectionnelle
}
```

`Customer` décrit une relation unidirectionnelle avec `Ticket` en utilisant la colonne de jointure `CUST_ID`.

2.2.5.3.2.3. Relation unidirectionnel avec une table de jointure

Une relation unidirectionnelle un à plusieurs avec une table de jointure est largement préférée. Cette association est décrite à travers l'annotation `@JoinTable`.

```
@Entity
public class Trainer {
    @OneToMany
    @JoinTable(
            name="TrainedMonkeys",
            joinColumns = { @JoinColumn( name="trainer_id") },
            inverseJoinColumns = @JoinColumn( name="monkey_id")
    )
    public Set<Monkey> getTrainedMonkeys() {
    ...
}

@Entity
public class Monkey {
    ... // pas de relation bidirectionnelle
}
```

`Trainer` décrit une relation unidirectionelle avec `Monkey` en utilisant la table de jointure `TrainedMonkeys`, avec une clef étrangère `trainer_id` vers `Trainer` (`joinColumns`) et une clef étrangère `monkey_id` vers `Monkey` (`inversejoinColumns`).

2.2.5.3.2.4. Valeurs par défaut

Si aucun mapping physique n'est déclaré, une relation unidirectionnelle un vers plusieurs utilise une table de jointure. Le nom de la table est la concaténation du nom de la table propriétaire, _, et le nom de la table de l'autre extrémité. Le nom des colonnes de la clef étrangère référençant la table propriétaire est la concaténation de la table propriétaire, _, et le nom des colonnes de la clef primaire. Le nom des colonnes de la clef étrangère référençant l'autre extrémité est la concaténation du nom de la propriété du propriétaire, _, et le nom des colonnes de la clef primaire de l'autre extrémité. Une contrainte d'unicité est ajoutée sur la clef étrangère référençant la table de l'autre extrémité pour réfléter le un à plusieurs.

```
@Entity
public class Trainer {
    @OneToMany
    public Set<Tiger> getTrainedTigers() {
    ...
}

@Entity
public class Tiger {
    ... // non bidirectionnelle
}
```

`Trainer` décrit une relation unidirectionnelle avec `Tiger` utilisant la table de jointure `Trainer_Tiger`, avec une clef étrangère `trainer_id` vers `Trainer` (nom de la table, _, identifiant de trainer) et une clef étrangère `trainedTigers_id` vers `Monkey` (nom de la propriété, _, colonne de la clef primaire de Tiger).

2.2.5.3.3. Many-to-many

2.2.5.3.3.1. Définition

Une association many-to-many est définie logiquement en utilisant l'annotation `@ManyToMany`. Vous devez aussi décrire la table d'association et les conditions de jointure en utilisant l'annotation `@JoinTable`. Si l'association est bidirectionnelle, une extrémité doit être la propriétaire et l'autre doit être marquée comme "inverse" (ie qu'elle sera ignorée lors de la mise à jour des valeurs de la relation dans la table d'association) :

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns={@JoinColumn(name="EMPER_ID")},
        inverseJoinColumns={@JoinColumn(name="EMPEE_ID")}
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade={CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy="employees"
        targetEntity=Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

Nous avons déjà montré les déclarations des relations "many" et détaillé les attributs de ces associations. Allons plus en profondeur dans la description de `@JoinTable` ; elle définit un `name`, un tableau de colonnes de jointure (un tableau dans une annotation est défini par {A, B, C}), et un tableau de colonnes de jointure inverse. Ces dernières sont les colonnes de la table d'association qui référencent la clef primaire de `Employee` ("l'autre extrémité").

Comme vu précédemment, l'autre extrémité ne doit pas décrire le mapping physique : un simple argument `mappedBy` contenant le nom de la propriété de l'extrémité propriétaire suffit à relier les deux.

2.2.5.3.3.2. Valeurs par défaut

Comme d'autres annotations, la plupart des valeurs d'une relation plusieurs à plusieurs sont inférées. Si aucun mapping physique n'est décrit dans une relation plusieurs à plusieurs unidirectionnelle, alors les règles suivantes s'appliquent. Le nom de la table est la concaténation du nom de la table propriétaire, _ et le nom de la table de l'autre extrémité. Le nom des colonnes de la clef étrangère référençant la table propriétaire est la concaténation du nom de la table propriétaire, _ et le nom des colonnes de la clef primaire de cette table. Le nom des colonnes de la clef étrangère référençant l'autre extrémité est la concaténation du nom de la propriété du propriétaire, _ et le nom des colonnes de la clef primaire de l'autre extrémité. Ce sont les mêmes règles que celles utilisées pour une relation un à plusieurs unidirectionnelle.

```

@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... // pas de relation bidirectionnelle
}
```

La table `Store_City` est utilisée comme table de jointure. La colonne `Store_id` est une clef étrangère vers la table `Store`. La colonne `implantedIn_id` est une clef étrangère vers la table `City`.

Si aucun mapping physique n'est décrit dans une relation plusieurs à plusieurs bidirectionnelle, alors les règles suivantes s'appliquent. Le nom de la table est la concaténation du nom de la table propriétaire, _ et le nom de la table de l'autre extrémité. Le nom des colonnes de la clef étrangère référençant la table propriétaire est la concaténation du nom de la propriété de l'autre extrémité, _ et le nom des colonnes de la clef primaire du propriétaire. Le nom des colonnes de la clef étrangère référençant l'autre extrémité est la concaténation du nom de la propriété du propriétaire, _ et le nom des colonnes de la clef primaire de l'autre extrémité. Ce sont les mêmes règles que celles utilisées pour une relation un à plusieurs unidirectionnelle.

```

```

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

La table `Store_Customer` est utilisée comme table de jointure. La colonne `stores_id` est une clef étrangère vers la table `Store`. La colonne `customers_id` est une clef étrangère vers la table `Customer`.

2.2.5.4. Persistance transitive avec les opérations en cascade

Vous avez probablement remarqué l'attribut `cascade` prenant comme valeur un tableau de `CascadeTypes`. Le concept de cascade dans EJB3 est similaire à la persistance transitive et les opérations en cascade dans Hibernate, mais avec une sémantique légèrement différente et les types de cascade suivants :

- CascadeType.PERSIST : effectue en cascade l'opération de persistance (création) sur les entités associées si persist() est appelée ou si l'entité est supervisée (par le gestionnaire d'entités)
- CascadeType.MERGE : effectue en cascade l'opération de fusion sur les entités associées si merge() est appélée ou si l'entité est supervisée
- CascadeType.REMOVE : effectue en cascade l'opération de suppression sur les entités associées si delete() est appelée
- CascadeType.REFRESH : effectue en cascade l'opération de rafraîchissement sur les entités associées si refresh() est appelée
- CascadeType.ALL : tous ceux du dessus

Veullez vous référer au chapitre 6.3 de la spécification EJB3 pour plus d'informations sur les opérations en cascade et la sémantique des opérations de création/fusion.

2.2.5.5. Récupération d'associations

Vous avez la possibilité de récupérer les entités associées soit immédiatement ("eager"), soit à la demande ("lazy"). Le paramètre `fetch` peut être positionné à `FetchType.LAZY` ou à `FetchType.EAGER`. `EAGER` essaiera d'utiliser une jointure externe pour rappatrier l'objet associé, alors que `LAZY` est la valeur par défaut et rapportera les données lorsque l'objet associé sera accédé pour la première fois. JPA-QL a aussi un mot clef `fetch` qui vous permet de surcharger le type de récupération pour une requête particulière. C'est très utile pour améliorer les performances et décider au cas par cas.

2.2.6. Mapper des clefs primaires et étrangères composées

Les clefs primaires composées utilisent une classe embarquée comme représentation de la clef primaire, donc vous devriez utiliser les annotations `@Id` et `@Embeddable`. Alternativement, vous pouvez utiliser l'annotation

`@EmbeddedId`. Notez que la classe dépendante doit être sérialisable et implementer `equals()`/`hashCode()`. Vous pouvez aussi utiliser `@IdClass` comme décrit dans Mapper des propriétés identifiantes.

```
@Entity
public class RegionalArticle implements Serializable {

    @Id
    public RegionalArticlePk getPk() { ... }
}

@Embeddable
public class RegionalArticlePk implements Serializable { ... }
```

ou alternativement

```
@Entity
public class RegionalArticle implements Serializable {

    @EmbeddedId
    public RegionalArticlePk getPk() { ... }
}

public class RegionalArticlePk implements Serializable { ... }
```

`@Embeddable` hérite le type d'accès de son entité d'appartenance à moins que l'annotation spécifique Hibernate `@AccessType` soit utilisée. Les clefs étrangères composées (si les valeurs par défaut ne sont pas utilisées) sont définies sur les associations en utilisant l'élément `@JoinColumns`, lequel est simplement un tableau de `@JoinColumns`. Il est considéré comme une bonne pratique d'exprimer `referencedColumnNames` explicitement. Sinon, Hibernate supposera que vous utilisez le même ordre de colonnes que dans la déclaration de la clef primaire.

```
@Entity
public class Parent implements Serializable {
    @Id
    public ParentPk id;
    public int age;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Set<Child> children; //unidirectionnelle
    ...
}
```

```
@Entity
public class Child implements Serializable {
    @Id @GeneratedValue
    public Integer id;

    @ManyToOne
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Parent parent; // unidirectionnelle
```

```
}
```

```
@Embeddable
public class ParentPk implements Serializable {
    String firstName;
    String lastName;
    ...
}
```

Notez l'usage explicite de `referencedColumnName`.

2.2.7. Mapper des tables secondaires

Vous pouvez mapper un simple entity bean vers plusieurs tables en utilisant les annotations de niveau classe `@SecondaryTable` ou `@SecondaryTables`. Pour dire qu'une colonne est dans une table particulière, utlisez le paramètre `table` de `@Column` ou `@JoinColumn`.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
```

Dans cet exemple, `name` sera dans `MainCat`. `storyPart1` sera dans `Cat1` et `storyPart2` sera dans `Cat2`. `Cat1` sera joint à `MainCat` avec `cat_id` comme clef étrangère, et `Cat2` avec `id` (ie le même nom de colonne que la colonne identifiante de `MainCat`). De plus, une contrainte d'unicité sur `storyPart2` a été renseignée.

Regardez le tutoriel EJB3 de JBoss ou la suite de tests unitaires d'Hibernate Annotations pour plus d'exemples.

2.3. Mapper des requêtes

2.3.1. Mapper des requêtes JPAQL/HQL

Vous pouvez mapper des requêtes JPA-QL/HQL en utilisant les annotations. `@NamedQuery` et `@NamedQueries` peuvent être définies au niveau de la classe ou dans un fichier JPA XML. Cependant, leurs définitions sont globales au scope de la session factory/entity manager factory. Une requête nommée est définie par son nom et la chaîne de caractères de la requête réelle.

```
<entity-mappings>
    <named-query name="plane.getAll">
        <query>select p from Plane p</query>
    </named-query>
    ...
</entity-mappings>
...

@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

Vous pouvez aussi fournir des indications de fonctionnement à une requête à travers un tableau de `QueryHints` avec l'attribut `hints`.

Les indications de fonctionnement Hibernate disponibles sont :

Tableau 2.2. Indications de fonctionnement d'une requête

Indication	description
org.hibernate.cacheable	Indique si la requête devrait interagir avec le cache de second niveau (par défaut à false)
org.hibernate.cacheRegion	Nom de la région du cache (si indéfinie, la valeur par défaut est utilisée)
org.hibernate.timeout	Timeout des requêtes
org.hibernate.fetchSize	Taille des result sets par fetch
org.hibernate.flushMode	Mode de flush utilisé pour cette requête
org.hibernate.cacheMode	Mode de cache utilisé pour cette requête
org.hibernate.readOnly	Indique si les entités chargées par cette requête devraient être en lecture seule ou pas (par défaut à

Indication	description
	false)
org.hibernate.comment	Commentaire de la requête, ajouté au SQL généré

2.3.2. Mapper des requêtes natives

Vous pouvez aussi mapper une requête native (ie une requête SQL). Pour ce faire, vous devez décrire la structure de l'ensemble de résultat SQL en utilisant `@SqlResultSetMapping` (ou `@SqlResultSetMappings` si vous prévoyez de définir plusieurs mappings de résultats). Comme `@NamedQuery`, un `@SqlResultSetMapping` peut être défini au niveau de la classe ou dans un fichier XML JPA. Cependant sa portée est globale à l'application.

Comme vous le verrez, un paramètre de `resultSetMapping` est défini dans `@NamedNativeQuery`, il représente le nom du `@SqlResultSetMapping` défini. Le mapping de l'ensemble des résultats déclare les entités récupérées par cette requête native. Chaque champ de l'entité est lié à un alias SQL (nom de colonne). Tous les champs de l'entité (dont ceux des classes filles) et les colonnes des clefs étrangères relatives aux entités doivent être présents dans la requête SQL. Les définitions des champs sont optionnelles, si elles ne sont pas fournies, elles mappent le même nom de colonne que celui déclaré sur la propriété de la classe.

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id", resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

Dans l'exemple ci-dessus, la requête nommée `night&area` utilise le mapping de résultats `joinMapping`. Ce mapping retourne 2 entités, `Night` et `Area`, chaque propriété est déclarée et associée à un nom de colonne, en fait le nom de colonne récupéré par la requête. Voyons maintenant une déclaration implicite de mapping propriété/colonne.

```
@Entity
@SqlResultSetMapping(name="implicit", entities=@EntityResult(entityClass=org.hibernate.test.annotation
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip", resultSetMapping="implicit"
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

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

```
    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

Dans cet exemple, nous décrivons seulement le membre de l'entité du mapping de résultats. Le mapping de propriété/colonne est fait en utilisant les valeurs de mapping de l'entité. Dans ce cas, la propriété `model` est liée à la colonne `model_txt`. Si l'association à une entité concernée implique une clef primaire composée, un élément `@FieldResult` devrait être utilisé pour chaque colonne de la clef étrangère. Le nom de `@FieldResult` est composé du nom de la propriété pour la relation, suivi par un point ("."), suivi par le nom ou le champ ou la propriété de la clef primaire.

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
    query="select name, model, speed, lname as lastn, fname as firstn, length, width, length * width a
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

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

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
        return captain;
    }
```

```
    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

Note

Si vous regardez la propriété dimension, vous verrez qu'Hibernate prend en charge la notation avec les points pour les objets embarqués (vous pouvez même avoir des objets embarqués imbriqués). Les implémentations EJB3 n'ont pas à prendre en charge cette fonctionnalité, mais nous le faisons :-)

Si vous récupérez une simple entité et si vous utilisez le mapping par défaut, vous pouvez utiliser l'attribut `resultClass` à la place de `resultSetMapping` :

```
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip",
    resultClass=SpaceShip.class)
public class SpaceShip {
```

Dans certaines de vos requêtes natives, vous devrez retourner des valeurs scalaires, par exemple lors de la construction de requêtes de rapport. Vous pouvez les mapper dans `@SqlResultsetMapping` avec `@ColumnResult`. En fait, vous pouvez même mélanger des retours d'entités et de valeurs scalaires dans la même requête native (ce n'est cependant probablement pas commun).

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar", query="select length*width as dimension from SpaceShip", resultSetMap
```

Une autre indication de fonctionnement spécifique aux requêtes natives a été présentée : `org.hibernate.callable` laquelle peut être à true ou à false fausse selon que la requête est une procédure stockée ou pas.

2.4. Extensions d'Hibernate Annotation

Hibernate 3.1 offre une variété d'annotations supplémentaires que vous pouvez mélanger/faire correspondre avec des entités EJB3. Elles ont été conçues comme une extension naturelle aux annotations EJB3.

Pour aller plus loin que les capacités d'EJB3, Hibernate fournit des annotations spécifiques qui correspondent aux fonctionnalités d'Hibernate. Le package `org.hibernate.annotations` contient toutes ces extensions d'annotations.

2.4.1. Entité

Vous pouvez finement paramétrer certaines des actions faites par Hibernate sur les entités au-delà de ce qu'offre la spécification EJB3.

`@org.hibernate.annotations.Entity` ajoute des méta-données supplémentaires qui peuvent être nécessaires au-delà de ce qui est défini dans l'annotation `@Entity` standard :

- mutable : indique si l'entité est modifiable ou non
- dynamicInsert : autorise le SQL dynamique pour les insertions
- dynamicUpdate : autorise le SQL dynamique pour les mise à jour
- selectBeforeUpdate : spécifie qu'Hibernate ne devrait jamais exécuter un UPDATE SQL à moins qu'il ne soit certain qu'un objet est réellement modifié
- polymorphism : indique si le polymorphisme d'entité est de type PolymorphismType.IMPLICIT (valeur par défaut) ou PolymorphismType.EXPLICIT
- persister : autorise la surcharge de l'implémentation de persistance fournie par défaut
- optimisticLock : stratégie de verrouillage optmiste (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY ou OptimisticLockType.ALL)

Note

@javax.persistence.Entity est encore obligatoire, @org.hibernate.annotations.Entity ne la remplace pas.

Voici quelques extensions d'annotations Hibernate supplémentaires.

`@org.hibernate.annotations.BatchSize` vous permet de définir la taille du batch lors de la récupération d'instances de cette entité (p. ex. `@BatchSize(size=4)`). Lors du chargement d'une entité donnée, Hibernate chargera alors toutes les entités non initialisées du même type dans le contexte de la persistance jusqu'à la taille du batch.

`@org.hibernate.annotations.Proxy` définit les attributs de chargement de l'entité. lazy (valeur par défaut) définit si la classe est chargée à la demande ou non. proxyClassName est l'interface utilisée pour générer le proxy (par défaut, la classe elle-même).

`@org.hibernate.annotations.Where` définit une clause WHERE SQL optionnelle utilisée lorsque des instances de cette classe sont récupérées.

`@org.hibernate.annotations.Check` déclare une contrainte de vérification optionnelle définie dans l'expression DDL.

`@OnDelete(action=OnDeleteAction.CASCADE)` sur des classes filles jointes : utilise une commande SQL DELETE en cascade lors de la suppression plutôt que le mécanisme habituel d'Hibernate.

`@Table(appliesTo="tableName", indexes = { @Index(name="index1", columnNames={"column1", "column2"} ) } )` crée les index définis sur les colonnes de la table `tableName`. Cela peut s'appliquer sur une table primaire ou une table secondaire. L'annotation `@Tables` vous permet d'avoir des index sur des tables différentes. Cette annotation est attendue là où `@javax.persistence.Table` ou `@javax.persistence.SecondaryTable`(s) sont déclarées.

Note

`@org.hibernate.annotations.Table` est un complément, pas un remplacement de `@javax.persistence.Table`. Surtout, si vous souhaitez changer le nom par défaut d'une table, vous devez utiliser `@javax.persistence.Table`, pas `@org.hibernate.annotations.Table`.

```
@Entity
@BatchSize(size=5)
@org.hibernate.annotations.Entity(
        selectBeforeUpdate = true,
        dynamicInsert = true, dynamicUpdate = true,
        optimisticLock = OptimisticLockType.ALL,
        polymorphism = PolymorphismType.EXPLICIT)
@Where(clause="1=1")
@org.hibernate.annotations.Table(name="Forest", indexes = { @Index(name="idx", columnNames = { "name",
public class Forest { ... }
```

```
@Entity
@Inheritance(
    strategy=InheritanceType.JOINED
)
public class Vegetable { ... }

@Entity
@OnDelete(action=OnDeleteAction.CASCADE)
public class Carrot extends Vegetable { ... }
```

2.4.2. Identifiant

`@org.hibernate.annotations.GenericGenerator` vous permet de définir un générateur d'identifiants Hibernate spécifique.

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {

@Id @GeneratedValue(generator="hibseq")
@GenericGenerator(name="hibseq", strategy = "seqhilo",
    parameters = {
        @Parameter(name="max_lo", value = "5"),
        @Parameter(name="sequence", value="heybabyhey")
    }
)
public Integer getId() {
```

`strategy` est le nom court de la stratégie du générateur Hibernate3 ou le nom pleinement qualifié de la classe d'une implémentation de `IdentifierGenerator`. Vous pouvez ajouter des paramètres avec l'attribut `parameters`.

Contrairement à son pendant standard, `@GenericGenerator` peut êtue utilisée dans les annotations au niveau du package, en faisant ainsi un générateur de niveau applicatif (comme s'il était dans un fichier JPA XML).

```
@GenericGenerator(name="hibseq", strategy = "seqhilo",
    parameters = {
        @Parameter(name="max_lo", value = "5"),
        @Parameter(name="sequence", value="heybabyhey")
    }
)
package org.hibernate.test.model
```

2.4.3. Propriété

2.4.3.1. Type d'accès

Le type d'accès est déduit de la position de `@Id` ou de `@EmbeddedId` dans la hiérarchie de l'entité. Les entités filles, les objets embarqués et les entités parentes mappés héritent du type d'accès de l'entité racine.

Dans Hibernate, vous pouvez surcharger le type d'accès pour :

- utiliser une stratégie d'accès personnalisée
- paramétrer finement le type d'accès au niveau de la classe ou au niveau de la propriété

Une annocation @AccessType a été présentée pour prendre en charge ce comportement. Vous pouvez définir le type d'accès sur :

- une entité
- une classe parente
- un objet embarqué
- une propriété

Le type d'accès est surchargé pour l'élément annoté, si surchargé sur une classe, toutes les propriétés de la classe donnée héritent du type d'accès. Pour les entités racines, le type d'accès est considéré par défaut comme celui de la hiérarchie entière (surchargeable au niveau de la classe ou de la propriété).

Si le type d'accès est marqué comme "propriété", les getters sont parcourus pour examiner les annotations, si le type d'accès est marqué comme "champ", ce sont les champs qui sont parcourus pour les annotations. Sinon les éléments marqués avec @Id ou @embeddedId sont scannés.

Vous pouvez surcharger une type d'accès pour une propriété, mais l'élément annoté ne sera pas influencé : par exemple, une entité ayant un type d'accès `field`, peut annoter un champ avec `@AccessType("property")`, le type d'accès sera alors "property" pour cet attribut, des annotations devront encore être portées sur les champs.

Si une classe parente ou un objet embarquable n'est pas annoté, le type d'accès de l'entité racine est utilisé (même si un type d'accès a été défini sur une classe parente ou un objet embarquable intermédiaire). Le principe de la poupée russe ne s'applique pas.

```
@Entity
public class Person implements Serializable {
    @Id  @GeneratedValue // type d'accès "champ"
    Integer id;

    @Embedded
    @AttributeOverrides({
    @AttributeOverride(name = "iso2", column = @Column(name = "bornIso2")),
    @AttributeOverride(name = "name", column = @Column(name = "bornCountryName"))
            })
    Country bornIn;
}

@Embeddable
@AccessType("property") // surcharge le type d'accès pour toutes les propriétés dans Country
public class Country implements Serializable {
    private String iso2;
    private String name;

    public String getIso2() {
        return iso2;
    }

    public void setIso2(String iso2) {
        this.iso2 = iso2;
    }

    @Column(name = "countryName")
    public String getName() {
        return name;
    }

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

2.4.3.2. Formule

Parfois, vous voulez effectuer certains calculs par la base de données plutôt que par la JVM, ou vous pourriez aussi vouloir créer une sorte de colonne virtuelle. Vous pouvez utilisez un fragment SQL (alias une formule) plutôt que de mapper un propriété sur une colonne. Cette sorte de propriété est en lecture seule (sa valeur est calculée par votre formule).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

Le fragment SQL peut être aussi complexe que vous le souhaitez, même avec des sous-selects inclus.

2.4.3.3. Type

`@org.hibernate.annotations.Type` surcharge le type Hibernate utilisé par défaut : ce n'est généralement pas nécessaire puisque le type est correctement inféré par Hibernate. Veuillez vous référer au guide de référence Hibernate pour plus d'informations sur les types Hibernate.

`@org.hibernate.annotations.TypeDef` et `@org.hibernate.annotations.TypeDefs` vous permettent de déclarer des définitions de type. Ces annotations sont placées au niveau de la classe ou du package. Notez que ces définitions seront globales pour la session factory (même au niveau de la classe) et que la définition du type doit être définie avant n'importe quelle utilisation.

```
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

...
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

Lors de l'utilisation d'un type utilisateur composé, vous devrez exprimer les définitions des colonnes. L'annotation `@Columns` a été mise en place dans ce but.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

2.4.3.4. Index

Vous pouvez définir un index sur une colonne particulière en utilisant l'annotation `@Index` sur une propriété d'une colonne, l'attribut columnNames sera ignoré.

```
@Column(secondaryTable="Cat1")
@Index(name="story1index")
public String getStoryPart1() {
    return storyPart1;
}
```

2.4.3.5. @Parent

A l'intérieur d'un objet embarquable, vous pouvez définir une des propriétés comme un pointeur vers l'élément propriétaire.

```
@Entity
public class Person {
    @Embeddable public Address address;
    ...
}

@Embeddable
public class Address {
    @Parent public Person owner;
    ...
}


person == person.address.owner
```

2.4.3.6. Propriétés générées

Certaines propriétés sont générées au moment de l'insertion ou de la mise à jour par votre base de données. Hibernate peut traiter de telles propriétés et déclencher un select subséquent pour lire ces propriétés.

```
@Entity
public class Antenna {
    @Id public Integer id;
    @Generated(GenerationTime.ALWAYS) @Column(insertable = false, updatable = false)
    public String longitude;

    @Generated(GenerationTime.INSERT) @Column(insertable = false)
    public String latitude;
}
```

Quand vous annotez votre propriété avec `@Generated`, vous devez vous assurer que l'insertion et la mise à jour n'entreront pas en conflit avec la stratégie de génération que vous avez choisie. Lorsque GenerationTime.INSERT est choisi, la propriété ne doit pas contenir de colonnes insérables ; lorsque GenerationTime.ALWAYS est choisi, la propriété ne doit pas contenir de colonnes qui puissent être insérées ou mises à jour.

Les propriétés `@Version` ne peuvent pas (par conception) être `@Generated(INSERT)`, elles doivent être `NEVER` ou `ALWAYS`.

2.4.4. Héritage

SINGLE_TABLE est une stratégie très puissante mais parfois, et surtout pour des systèmes pré-existants, vous ne pouvez pas ajouter une colonne discriminante supplémentaire. Pour cela Hibernate a mis en place la notion de formule discriminante : `@DiscriminatorFormula` est une remplaçcant de `@DiscriminatorColumn` et utilise un fragment SQL en tant que formule pour la résolution du discriminant (pas besoin d'avoir une colonne dédiée).

```
@Entity
@DiscriminatorForumla("case when forest_type is null then 0 else forest_type end")
public class Forest { ... }
```

Par défaut, lors du requêtage sur les entités les plus hautes, Hibernate ne met pas de restriction sur la colonne discriminante. Ceci peut être un inconvénient si cette colonne contient des valeurs qui ne sont pas mappées dans votre hiérarchie (avec `@DiscriminatorValue`). Pour contourner ca, vous pouvez utilser `@ForceDiscriminator` (au niveau de la classe, à côté de `@DiscriminatorColumn`). Hibernate listera alors les

valeurs disponibles lors du chargement des entités.

2.4.5. Annotations concernant les simples associations

Par défaut, lorsqu'Hibernate ne peut pas résoudre l'association parce que l'élément associé attendu n'est pas dans la base de données (mauvais identifiant sur la colonne de l'association), une exception est levée par Hibernate. Cela pourrait être un inconvénient pour des schémas pré-existants et mal maintenus. Vous pouvez demander à Hibernate d'ignorer de tels éléments plutôt que de lever une exception en utilisant l'annotation `@NotFound`. Cette annotation peut être utilisée sur une association `@OneToOne` (avec une clef étrangère), `@ManyToOne`, `@OneToMany` ou `@ManyToMany`.

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Parfois vous voulez déléguer à votre base de données la suppression en cascade lorsqu'une entité donnée est supprimée.

```
@Entity
public class Child {
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

Dans ce cas, Hibernate génère une contrainte de suppression en cascade au niveau de la base de données.

Les contraintes de clef étrangère, bien que générées par Hibernate, ont un nom justement illisible. Vous pouvez surcharger le nom de la contrainte par l'utilisation de `@ForeignKey`.

```
@Entity
public class Child {
    ...
    @ManyToOne
    @ForeignKey(name="FK_PARENT")
    public Parent getParent() { ... }
    ...
}

alter table Child add constraint FK_PARENT foreign key (parent_id) references Parent
```

2.4.5.1. Options de chargement et modes de récupération

EJB3 arrive avec l'option `fetch` pour définir le chargement à la demande et les modes de récupération, cependant Hibernate a beaucoup plus d'options dans ce domaine. Pour finement paramétrer le chargement à la demande et les stratégies de récupération, quelques annotations supplémentaires ont été mises en place :

- `@LazyToOne` : définit l'option de chargement à la demande sur les associations `@ManyToOne` et `@OneToOne`. `LazyToOneOption` peut être `PROXY` (ie utiliser un chargement à la demande basé sur un proxy), `NO_PROXY` (utilise un chargement à la demande sur l'ajout de bytecode - notez qu'un temps de construction du bytecode est nécessaire) et `FALSE` (association sans chargement à la demande) ;

- `@LazyCollection` : définit l'option de chargement à la demande sur les associations `@ManyToMany` et `@OneToMany`. LazyCollectionOption peut être `TRUE` (la collection est chargée à la demande lorsque son état est accédé), `EXTRA` (la collection est chargée à la demande et toutes les opérations essaieront d'éviter le chargement de la collection, c'est surtout utile pour de grosses collections lorsque le chargement de tous les éléments n'est pas nécessaire) et `FALSE` (association sans chargement à la demande) ;

- `@Fetch` : définit une stratégie de récupération utilisée pour charger l'association. `FetchMode` peut être `SELECT` (un select est déclenché lorsque l'association a besoin d'être chargée), `SUBSELECT` (disponible uniquement pour des collections, utilise une stratégie de sous select - veuillez vous référer à la documentation de référence d'Hibernate pour plus d'informations) ou `JOIN` (utilise un JOIN SQL pour charger l'association lors du chargement de l'entité propriétaire). `JOIN` surcharge n'importe quel attribut de chargement à la demande (une association chargée avec la stratégie `JOIN` ne peut pas être chargée à la demande).

Les annotations Hibernate surchargent les options de récupération EJB3.

Tableau 2.3. Chargement à la demande et options de récupération équivalentes

Annotations	Chargement à la demande	Récupération
@[One|Many]ToOne](fetch=FetchType.LAZY)	@LazyToOne(PROXY)	@Fetch(SELECT)
@[One|Many]ToOne](fetch=FetchType.EAGER)	@LazyToOne(FALSE)	@Fetch(JOIN)
@ManyTo[One|Many](fetch=FetchType.LAZY)	@LazyCollection(TRUE)	@Fetch(SELECT)
@ManyTo[One|Many](fetch=FetchType.EAGER)	@LazyCollection(FALSE)	@Fetch(JOIN)

2.4.6. Annotations concernant les collections

2.4.6.1. Améliorer les configurations des collections

Il est possible de configurer :

- la taille des batchs pour les collections en utilisant @BatchSize

- la clause where, en utilisant @Where (appliquée à l'entité cible) ou @WhereJoinTable (appliquée à la table de l'association)

- la clause de vérification, en utilsant @Check

- la clause SQL order by, en utilisant @OrderBy

- la stratégie de suppression en cascade avec @OnDelete(action=OnDeleteAction.CASCADE)

Vous pouvez aussi déclarer un comparateur de tri, utilisez l'annotation `@Sort`. Exprimez le type de comparateur que vous voulez entre "non trié" (NdT : unsorted), "ordre naturel" (NdT : natural) ou un comparateur personnalisé. Si vous voulez utilisez votre propre implémentation de comparateur, vous devrez indiquer la classe d'implémentation en utilisant l'attribut `comparator`. Notez que vous avez besoin d'utiliser l'interface

`SortedSet` ou `SortedMap`.

```
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    @Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
    @Where(clause="1=1")
    @OnDelete(action=OnDeleteAction.CASCADE)
    public SortedSet<Ticket> getTickets() {
        return tickets;
    }
```

Veuillez vous référer aux descriptions précédentes de ces annotations pour plus d'informations.

Les contraintes de clef étrangère, bien que générées par Hibernate, ont un nom illisible. Vous pouvez surcharger le nom de la contrainte en utilisant `@ForeignKey`. Notez que cette annotation doit être placée du côté possédant la relation, `inverseName` référençant la contrainte de l'autre côté.

```
@Entity
public class Woman {
    ...
    @ManyToMany(cascade = {CascadeType.ALL})
    @ForeignKey(name = "TO_WOMAN_FK", inverseName = "TO_MAN_FK")
    public Set<Man> getMens() {
        return mens;
    }
}

alter table Man_Woman add constraint TO_WOMAN_FK foreign key (woman_id) references Woman
alter table Man_Woman add constraint TO_MAN_FK foreign key (man_id) references Man
```

2.4.6.2. Types de collection extra

2.4.6.2.1. List

Outre EJB3, Hibernate Annotations prend en charge les véritables `List` et `Array`. Mappez votre collection de la même manière que d'habitude et ajoutez l'annotation `@IndexColumn`. Cette annotation vous permet de décrire la colonne qui contiendra l'index. Vous pouvez aussi déclarer la valeur de l'index en base de données qui représente le premier élément (alias index de base). La valeur habituelle est `0` ou `1`.

```
@OneToMany(cascade = CascadeType.ALL)
@IndexColumn(name = "drawer_position", base=1)
public List<Drawer> getDrawers() {
    return drawers;
}
```

> **Note**
>
> Si vous oubliez de positionner `@IndexColumn`, la sémantique du bag est appliquée. Si vous voulez la sémantique du bag sans ses limitations, considérez l'utilisation de `@CollectionId`.

2.4.6.2.2. Map

Hibernate Annotations prend aussi en charge le mapping de véritables Maps, si `@javax.persistence.MapKey` n'est pas positionnée, Hibernate mappera l'élément clef ou l'objet embarquable dans ses propres colonnes. Pour surcharger les colonnes par défaut, vous pouvez utiliser `@org.hibernate.annotations.MapKey` si votre clef est un type de base (par défaut à `mapkey`) ou un objet embarquable, ou vous pouvez utiliser `@org.hibernate.annotations.MapKeyManyToMany` si votre clef est une entité.

2.4.6.2.3. Associations bidirectionnelle avec des collections indexées

Une association bidirectionnelle où une extrémité est représentée comme une `@IndexColumn` ou une `@org.hibernate.annotations.MapKey[ManyToMany]` requiert une considération spéciale. S'il y a une propriété de la classe enfante qui mappe la colonne de l'index, pas de problème, nous pouvons continuer en utilisant `mappedBy` sur le mapping de la collection :

```
@Entity
public class Parent {
    @OneToMany(mappedBy="parent")
    @org.hibernate.annotations.MapKey(columns=@Column(name="name"))
    private Map<String, Child> children;
    ...
}

@Entity
public class Parent {
    ...
    @Basic
    private String name;

    @ManyToOne
    @JoinColumn(name="parent_id", nullable=false)
    private Parent parent;
    ...
}
```

Mais s'il n'y a pas de telle propriété sur la classe enfante, nous ne pouvons pas penser que l'association est vraiment bidirectionnelle (il y a des informations disponibles à une extrémité qui ne sont pas disponibles à l'autre). Dans ce cas, nous ne pouvons pas mapper la collection avec `mappedBy`. A la place, nous pourrions utiliser le mapping suivant :

```
@Entity
public class Parent {
    @OneToMany
    @org.hibernate.annotations.MapKey(columns=@Column(name="name"))
    @JoinColumn(name="parent_id", nullable=false)
    private Map<String, Child> children;
    ...
}

@Entity
public class Parent {
    ...
    @ManyToOne
    @JoinColumn(name="parent_id", insertable=false, updatable=false, nullable=false)
    private Parent parent;
    ...
}
```

Notez que dans ce mapping, l'extrémité de l'association dont la valeur est une collection est responsable des mises à jour pour la clef étrangère.

2.4.6.2.4. Bag avec une clef primaire

Une autre fonctionnalité intéressante est la possibilité de définir une clef primaire subrogée à une collection bag. Ceci enlève pas mal d'inconvénients des bags : mise à jour et suppression sont efficaces, plus d'un bag `EAGER` par requête ou par entité. Cette clef primaire sera contenue dans une colonne supplémentaire de votre table de collection mais ne sea pas visible par l'application Java. @CollectionId est utilisée pour marquer une collection comme "id bag", ca permet aussi de surcharger les colonnes de la clef primaire, le type de la clef primaire et la stratégie du générateur. La stratégie peut être `identity`, ou n'importe quel nom de générateur défini de votre application.

```
@Entity
```

```
@TableGenerator(name="ids_generator", table="IDS")
public class Passport {
    ...

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name="PASSPORT_VISASTAMP")
    @CollectionId(
        columns = @Column(name="COLLECTION_ID"),
        type=@Type(type="long"),
        generator = "ids_generator"
    )
    private Collection<Stamp> visaStamp = new ArrayList();
    ...
}
```

2.4.6.2.5. Collection d'éléments ou d'éléments composés

Hibernate Annotations prend aussi en charge les collections de types core (Integer, String, Enums, ...), les collections d'objets embarquables et même les tableaux de types primitifs. Ce sont des collections d'éléments.

Une collection d'éléments doit être annotée comme `@CollectionOfElements` (en tant que remplaçant de `@OneToMany`). Pour définir la table de la collection, l'annotation `@JoinTable` est utilisée sur la propriété de l'association, `joinColumns` définit les colonnes de jointure entre la table de l'entité primaire et la table de la collection (inverseJoincolumn est inutile et devrait être laissé à vide). Pour une collection de types core ou un tableau de types primitifs, vous pouvez surcharger la définition de la colonne de l'élément en utilisant `@Column` sur la propriété de l'association. Vous pouvez aussi surcharger les colonnes d'une collection d'objets embarquables en utilisant `@AttributeOverride`. Pour atteindre l'élément de la collection, vous avez besoin d'ajouter "element" au nom de l'attribut surchargé (p. ex. "element" pour les types core, ou "element.serial" pour la propriété serial d'un élément embarqué). Pour atteindre l'index/clef d'une collection, ajoutez "key" à la place.

```
@Entity
public class Boy {
    private Integer id;
    private Set<String> nickNames = new HashSet<String>();
    private int[] favoriteNumbers;
    private Set<Toy> favoriteToys = new HashSet<Toy>();
    private Set<Character> characters = new HashSet<Character>();

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    @CollectionOfElements
    public Set<String> getNickNames() {
        return nickNames;
    }

    @CollectionOfElements
    @JoinTable(
            table=@Table(name="BoyFavoriteNumbers"),
            joinColumns = @JoinColumn(name="BoyId")
    )
    @Column(name="favoriteNumber", nullable=false)
    @IndexColumn(name="nbr_index")
    public int[] getFavoriteNumbers() {
        return favoriteNumbers;
    }

    @CollectionOfElements
    @AttributeOverride( name="element.serial", column=@Column(name="serial_nbr") )
    public Set<Toy> getFavoriteToys() {
        return favoriteToys;
    }
```

```
    @CollectionOfElements
    public Set<Character> getCharacters() {
        return characters;
    }
    ...
}

public enum Character {
    GENTLE,
    NORMAL,
    AGGRESSIVE,
    ATTENTIVE,
    VIOLENT,
    CRAFTY
}

@Embeddable
public class Toy {
    public String name;
    public String serial;
    public Boy owner;

    public String getName() {
        return name;
    }

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

    public String getSerial() {
        return serial;
    }

    public void setSerial(String serial) {
        this.serial = serial;
    }

    @Parent
    public Boy getOwner() {
        return owner;
    }

    public void setOwner(Boy owner) {
        this.owner = owner;
    }

    public boolean equals(Object o) {
        if ( this == o ) return true;
        if ( o == null || getClass() != o.getClass() ) return false;

        final Toy toy = (Toy) o;

        if ( !name.equals( toy.name ) ) return false;
        if ( !serial.equals( toy.serial ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = name.hashCode();
        result = 29 * result + serial.hashCode();
        return result;
    }
}
```

Sur une collection d'objets embarquables, l'objet embarquable peut avoir une propriété annotée avec `@Parent`. Cette propriété pointera alors vers l'entité contenant la collection.

Note

Les versions précédentes d'Hibernate Annotations utilisaient `@OneToMany` pour marquer une collection d'éléments. Suite à des incohérences sémantiques, nous avons mis en place l'annotation `@CollectionOfElements`. Pour marquer des collections d'éléments, l'ancienne façon fonctionne encore mais elle est considérée comme "deprecated" et ne sera plus prise en charge dans les futures versions.

2.4.7. Cache

Pour optimiser vos accès à la base de données, vous pouvez activer le cache de second niveau d'Hibernate. Ce cache est configurable par entité et par collection.

`@org.hibernate.annotations.Cache` définit la stratégie de cache et la région du cache de second niveau donné. Cette annotation peut être appliquée à une entité racine (pas les entités filles), et sur les collections.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

```
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    public SortedSet<Ticket> getTickets() {
        return tickets;
    }
```

```
@Cache(
    CacheConcurrencyStrategy usage();               (1)
    String region() default "";                     (2)
    String include() default "all";                 (3)
)
```

(1) usage : la stratégie de concurrence du cache donné (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL) ;

(2) region (optionnel) : la région du cache (par défaut le nom complet de la classe avec le nom du package, ou le nom complet du rôle de la collection) ;

(3) `include` (optionnel) : "all" pour inclure toutes les propriétés, "non-lazy" pour inclure seulement les propriétés qui ne sont pas chargées à la demande (valeur par défaut : all).

2.4.8. Filtres

Hibernate a la capacité d'appliquer des filtres arbitraires à la partie supérieure de vos données. Ces filtres sont appliqués au moment de l'exécution sur une session donnée. Vous avez tout d'abord besoin de les définir.

`@org.hibernate.annotations.FilterDef` ou `@FilterDefs` déclarent des définitions de filtre utilisées par les filtres ayant le même nom. Une définition de filtre a un name() et un tableau de parameters(). Un paramètre vous permettra d'ajuster le comportement du filtre au moment de l'exécution. Chaque paramètre est défini par une `@ParamDef` qui a un nom et un type. Vous pouvez aussi définir un paramètre defaultCondition() pour une `@ParamDef` donnée pour positionner la condition par défaut à utiliser lorsqu'aucune n'est définie dans chaque `@Filter` individuelle. Une `@FilterDef` peut être définie au niveau de la classe ou du package.

Nous avons besoin de définir la clause du filtre SQL appliqué au chargement de l'entité ou au chargement de la collection. `@Filter` est utilisée et placée sur l'entité ou l'élément de la collection.

```
@Entity
@FilterDef(name="minLength", parameters={ @ParamDef( name="minLength", type="integer" ) } )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

Lorsque la collection utilise une table d'association comme représentation relationnelle, vous pourriez vouloir appliquer la condition du filtre à la table de l'association elle-même ou à la table de l'entité cible. Pour appliquer la contrainte sur l'entité cible, utilisez l'annotation habituelle `@Filter`. Cependant, si vous voulez ciblez la table d'association, utilisez l'annotation `@FilterJoinTable`.

```
    @OneToMany
    @JoinTable
    // filtre sur la table de l'entité cible
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    // filtre sur la table d'association
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

2.4.9. Requête

Puisqu'Hibernate a plus de fonctionnalités sur les requêtes nommées que définies dans la spécification EJB3, `@org.hibernate.annotations.NamedQuery`, `@org.hibernate.annotations.NamedQueries`, `@org.hibernate.annotations.NamedNativeQuery` et `@org.hibernate.annotations.NamedNativeQueries` ont été mis en place. Elles ajoutent des attributs à la version standard et peuvent être utilisées comme remplaçant :

- flushMode : définit le mode de flush de la requête (Always, Auto, Commit ou Never)
- cacheable : indique si la requête devrait être cachée ou non
- cacheRegion : région du cache utilisée si la requête est cachée
- fetchSize : taille de l'expression de récupération JDBC pour cette requête
- timeout : timeout de la requête
- callable : pour les requêtes natives seulement, mettre à true pour les procédures stockées
- comment : si les commentaires sont activés, le commentaire vu lorsque la requête est envoyée vers la base de données
- cacheMode : mode d'intéraction du cache (get, ignore, normal, put ou refresh)
- readOnly : indique si les éléments récupérés à partir de la requête sont en lecture seule ou pas

Ces indications de fonctionnement peuvent être positionnées sur les annotations standards `@javax.persistence.NamedQuery` avec l'annotation `@QueryHint`. Un autre avantage clef est la possibilité de positionner ces annotations au niveau du package.

Chapitre 3. Surcharger des méta-données à travers du XML

La cible primaire pour les méta-données dans EJB3 sont les annotations, mais la spécification EJB3 fournit un moyen de surcharger ou remplacer les méta-données définies par des annotations à travers un descripteur de déploiement XML. Dans la version courante, seule la surcharge des annotations pure EJB3 est prise en charge. Si vous souhaitez utiliser des caractéristiques spécifiques à Hibernate dans des entités, vous devrez utiliser les annotations ou vous replier sur les fichiers hbm. Vous pouvez bien sûr mélanger et faire correspondre des entités annotées et des entités décrites dans des fichiers hbm.

La suite de test unitaires montre des exemples supplémentaires de fichier XML.

3.1. Principes

La structure du descripteur de déploiement XML a été conçue pour refléter celle des annotations. Donc si vous connaissez la structure des annotations, utiliser le schéma XML sera facile pour vous.

Vous pouvez définir un ou plusieurs fichiers XML décrivant vos méta-données, ces fichiers seront fusionnés par le moteur de surcharge.

3.1.1. Méta-données de niveau global

Vous pouvez définir des méta-données de niveau global disponibles pour tous les fichiers XML. Vous ne devez pas définir ces méta-données plus d'une fois par déploiement.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

    <persistence-unit-metadata>
        <xml-mapping-metadata-complete/>
        <persistence-unit-defaults>
            <schema>myschema</schema>
            <catalog>mycatalog</catalog>
            <cascade-persist/>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
```

`xml-mapping-metadata-complete` signifie que toutes les entités, classes mères mappées et méta-données devraient être récupérées à partir du XML (c'est-à-dire ignorer les annotations).

`schema / catalog` surchargera toutes les définitions par défaut de schéma et de catalogue dans les méta-données (XML et annotations).

`cascade-persist` signifie que toutes les associations ont PERSIST comme type de cascade. Nous vous recommandons de ne pas utiliser cette fonctionnalité.

3.1.2. Méta-données de niveau entité

Vous pouvez définir ou surcharger des informations de méta-données sur une entité donnée.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings                                                                (1)
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

    <package>org.hibernate.test.reflection.java.xml</package>                   (2)
    <entity class="Administration" access="PROPERTY" metadata-complete="true">  (3)
        <table name="tbl_admin">                                                (4)
            <unique-constraint>
                <column-name>firstname</column-name>
                <column-name>lastname</column-name>
            </unique-constraint>
        </table>
        <secondary-table name="admin2">                                         (5)
            <primary-key-join-column name="admin_id" referenced-column-name="id"/>
            <unique-constraint>
                <column-name>address</column-name>
            </unique-constraint>
        </secondary-table>
        <id-class class="SocialSecurityNumber"/>                                (6)
        <inheritance strategy="JOINED"/>                                        (7)
        <sequence-generator name="seqhilo" sequence-name="seqhilo"/>            (8)
        <table-generator name="table" table="tablehilo"/>                       (9)
        ...
    </entity>

    <entity class="PostalAdministration">
        <primary-key-join-column name="id"/>                                    (10)
        ...
    </entity>
</entity-mappings>
```

(1) `entity-mappings` : entity-mappings est l'élément racine pour tous les fichiers XML. Vous devez déclarer le schéma xml, le fichier du schéma est inclus dans le fichier hibernate-annotations.jar, aucun accès à internet ne sera effectué par Hibernate Annotations.

(2) `package` (optionnel) : package par défaut utilisé pour tous les noms de classes sans package dans le fichier de descripteur de déploiement donné.

(3) `entity` : décrit une entité.

`metadata-complete` définit si la description des méta-données pour cet élément est complète ou pas (en d'autres mots, si les annotations présentes au niveau de la classe devraient être prises en compte ou pas).

Une entité doit avoir un attribut `class` référençant une classe java à laquelle s'applique les méta-données.

Vous pouvez surcharger un nom d'entité avec l'attribut `name`, si aucun n'est défini et si une annotation `@Entity.name` est présente, alors elle est utilisée (et établit que les méta-données ne sont pas complètes).

Pour un élément avec des méta-données complètes (voir ci-dessous), vous pouvez définir un attribut `access` (soit `FIELD`, soit `PROPERTY` (valeur par défaut)). Pour un élément avec des méta-données incomplètes, si `access` n'est pas défini, la position de @Id permettra de le déterminer, si `access` est défini, sa valeur est utilisée.

(4) `table` : vous pouvez déclarer des propriétés de table (nom, schéma, catalogue), si aucune n'est définie, l'annotation java est utilisée.

Vous pouvez définir une ou plusieurs contraintes d'unicité comme dans l'exemple.

(5) `secondary-table` : définit une table secondaire très semblable à une table habituelle excepté que vous pouvez définir les colonnes de clef primaire / clef étrangère avec l'élément `primary-key-join-column`.

Sur des méta-données incomplètes, les annotations de table secondaire sont utilisées seulement s'il n'y a pas de `secondary-table` de défini, sinon les annotations sont ignorées.

(6) `id-class` : définit la classe identifiante comme le fait `@IdClass`.

(7) `inheritance` : définit la stratégie d'héritage (`JOINED`, `TABLE_PER_CLASS`, `SINGLE_TABLE`) ; disponible seulement au niveau de l'élément racine.

(8) `sequence-generator` : définit un générateur de séquence.

(9) `table-generator` : définit un générateur de table.

(10) `primary-key-join-column` : définit la colonne de jointure sur la clef primaire pour les entités filles lorsque la stratégie d'héritage utilisée est JOINED.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

    <package>org.hibernate.test.reflection.java.xml</package>
    <entity class="Music" access="PROPERTY" metadata-complete="true">
        <discriminator-value>Generic</discriminator-value>                          (1)
        <discriminator-column length="34"/>
        ...
    </entity>

    <entity class="PostalAdministration">
        <primary-key-join-column name="id"/>
        <named-query name="adminById">                                              (2)
            <query>select m from Administration m where m.id = :id</query>
            <hint name="org.hibernate.timeout" value="200"/>
        </named-query>
        <named-native-query name="allAdmin" result-set-mapping="adminrs">           (3)
            <query>select *, count(taxpayer_id) as taxPayerNumber
            from Administration, TaxPayer
            where taxpayer_admin_id = admin_id group by ...</query>
            <hint name="org.hibernate.timeout" value="200"/>
        </named-native-query>
        <sql-result-set-mapping name="adminrs">                                     (4)
            <entity-result entity-class="Administration">
                <field-result name="name" column="fld_name"/>
            </entity-result>
            <column-result name="taxPayerNumber"/>
        </sql-result-set-mapping>
        <attribute-override name="ground">                                          (5)
            <column name="fld_ground" unique="true" scale="2"/>
        </attribute-override>
        <association-override name="referer">
            <join-column name="referer_id" referenced-column-name="id"/>
        </association-override>
        ...
    </entity>
</entity-mappings>
```

(1) `discriminator-value / discriminator-column` : définissent la colonne et la valeur discriminantes lorsque la stratégie d'héritage choisie est SINGLE_TABLE.

(2) `named-query` : définit les requêtes nommées et potentiellement les indices qui leur sont associés. Ces définitions sont ajoutées à celles définies dans les annotations, si deux définitions ont le même nom, la version XML a la priorité.

(3) `named-native-query` : définit une requête SQL nommée et le mapping de son résultat. Alternativement, vous pouvez définir `result-class`. Ces définitions sont ajoutées à celles definies dans les annotations, si deux définitions ont le même nom, la version XML a la priorité.

(4) `sql-result-set-mapping` : décrit la structure du mapping des résultats. Vous pouvez définir des mappings de colonnes et d'entité. Ces définitions sont ajoutées à celles définies dans les annotations, si deux définitions ont le même nom, la version XML a la priorité.

(5) `attribute-override / association-override` : surcharge la définition d'une colonne ou d'une colonne de jointure. Cette surcharge est ajoutée à celle définie dans les annotations.

La même chose s'applique à `<embeddable>` et `<mapped-superclass>`.

3.1.3. Méta-données de niveau propriété

Vous pouvez bien sûr définir des surcharges XML pour des propriétés. Si les méta-données sont définies comme incomplètes, alors les propriétés supplémentaires (c'est-à-dire au niveau Java) seront ignorées. Toutes les méta-données de niveau propriété sont définies par `entity/attributes`, `mapped-superclass/attributes` ou `embeddable/attributes`.

```
    <attributes>
        <id name="id">
            <column name="fld_id"/>
            <generated-value generator="generator" strategy="SEQUENCE"/>
            <temporal>DATE</temporal>
            <sequence-generator name="generator" sequence-name="seq"/>
        </id>
        <version name="version"/>
        <embedded name="embeddedObject">
            <attribute-override name"subproperty">
                <column name="my_column"/>
            </attribute-override>
        </embedded>
        <basic name="status" optional="false">
            <enumerated>STRING</enumerated>
        </basic>
        <basic name="serial" optional="true">
            <column name="serialbytes"/>
            <lob/>
        </basic>
        <basic name="terminusTime" fetch="LAZY">
            <temporal>TIMESTAMP</temporal>
        </basic>
    </attributes>
```

Vous pouvez surcharger une propriété avec `id`, `embedded-id`, `version`, `embedded` et `basic`. Chacun de ces éléments peuvent avoir des sous-éléments : `lob`, `temporal`, `enumerated`, `column`.

3.1.4. Méta-données au niveau association

Vous pouvez définir des surcharges XML pour les associations. Toutes les méta-données de niveau association sont définies par `entity/attributes`, `mapped-superclass/attributes` ou `embeddable/attributes`.

```
    <attributes>
        <one-to-many name="players" fetch="EAGER">
            <map-key name="name"/>
            <join-column name="driver"/>
            <join-column name="number"/>
        </one-to-many>
        <many-to-many name="roads" target-entity="Administration">
            <order-by>maxSpeed</order-by>
            <join-table name="bus_road">
                <join-column name="driver"/>
                <join-column name="number"/>
                <inverse-join-column name="road_id"/>
                <unique-constraint>
                    <column-name>driver</column-name>
                    <column-name>number</column-name>
                </unique-constraint>
            </join-table>
        </many-to-many>
```

```
        <many-to-many name="allTimeDrivers" mapped-by="drivenBuses">
    </attributes>
```

Vous pouvez surcharger une association avec `one-to-many`, `one-to-one`, `many-to-one`, et `many-to-many`. Chacun de ces éléments peut avoir des sous-éléments : `join-table` (qui peut avoir des `join-columns` et des `inverse-join-columns`), `join-columns`, `map-key`, et `order-by`. `mapped-by` et `target-entity` peuvent être définis en tant qu'attributs lorsque cela a du sens. Une fois de plus la structure est le reflet de la structure des annotations. Vous pouvez trouver toutes les informations de sémantique dans le chapitre décrivant les annotations.

Chapitre 4. Hibernate Validator

Les annotations sont une manière très commode et élégante pour spécifier des contraintes invariantes sur un modèle de données. Vous pouvez, par exemple, indiquer qu'une propriété ne devrait pas être nulle, que le solde d'un compte devrait être strictement positif, etc. Ces contraintes de modèle de données sont déclarées dans le bean lui-même en annotant ses propriétés. Un validateur peut alors les lire et vérifier les violations de contraintes. Le mécanisme de validation peut être exécuté dans différentes couches de votre application (présentation, accès aux données) sans devoir dupliquer ces règles. Hibernate Validator a été conçu dans ce but.

Hibernate Validator fonctionne sur deux niveaux. D'abord, il est capable de vérifier des violations de contraintes sur les instances d'une classe en mémoire. Ensuite, il peut appliquer les contraintes au méta-modèle d'Hibernate et les incorporer au schéma de base de données généré.

Chaque annotation de contrainte est associée à l'implémentation du validateur responsable de vérifier la contrainte sur l'instance de l'entité. Un validateur peut aussi (optionnellement) appliquer la contrainte au méta-modèle d'Hibernate, permettant à Hibernate de générer le DDL qui exprime la contrainte. Avec le listener d'événements approprié, vous pouvez exécuter l'opération de vérification lors des insertions et des mises à jour effectuées par Hibernate. Hibernate Validator n'est pas limité à Hibernate. Vous pouvez facilement l'utiliser n'importe où dans votre application.

Lors de la vérification des instances à l'exécution, Hibernate Validator retourne des informations à propos des violations de contraintes dans un tableau de `InvalidValues`. Parmi d'autres informations, `InvalidValue` contient un message de description d'erreur qui peut inclure les valeurs des paramètres associés à l'annotation (p. ex. la limite de taille), et des chaînes de caractères qui peuvent être externalisées avec un `ResourceBundle`.

4.1. Contraintes

4.1.1. Qu'est-ce qu'une contrainte ?

Une contrainte est représentée par une annotation. Une contrainte a généralement des attributs utilisés pour paramétrer les limites des contraintes. La contrainte s'applique à l'élément annoté.

4.1.2. Contraintes intégrées

Hibernate Validator arrive avec des contraintes intégrées, lesquelles couvrent la plupart des vérifications de données de base. Comme nous le verrons plus tard, vous n'êtes pas limité à celles-ci, vous pouvez écrire vos propres contraintes en une minute.

Tableau 4.1. Contraintes intégrées

Annotation	S'applique à	Vérification à l'exécution	Impact sur les méta-données d'Hibernate
@Length(min=, max=)	propriété (String)	vérifie si la longueur de la chaîne de caractères est comprise dans l'intervalle	la longueur de la colonne sera positionnée à max
@Max(value=)	propriété (nombre ou chaîne de caractères représentant un nombre)	vérifie si la valeur est inférieure ou égale à max	ajoute une contrainte de vérification sur la colonne

Annotation	S'applique à	Vérification à l'exécution	Impact sur les méta-données d'Hibernate
@Min(value=)	propriété (nombre ou chaîne de caractères représentant un nombre)	vérifie si la valeur est supérieure ou égale à max	ajoute une contrainte de vérification sur la colonne
@NotNull	propriété	vérifie si la valeur n'est pas nulle	les colonnes sont marquées "not null"
@Past	propriété (Date ou Calendar)	vérifie si la date est dans le passé	ajoute une contrainte de vérification sur la colonne
@Future	propriété (Date ou Calendar)	vérifie si la date est dans le futur	aucun
@Pattern(regex="regexp", flag=)	propriété (String)	vérifie si la propriété correspond à l'expression rationnelle donnée (pour "flag", voir `java.util.regex.Pattern`)	aucun
@Range(min=, max=)	propriété (nombre ou chaîne de caractères représentant un nombre)	vérifie si la valeur est comprise entre min et max (inclus)	ajoute une contrainte de vérification sur la colonne
@Size(min=, max=)	propriété (tableau, collection, map)	vérifie si la taille de l'élément est comprise entre min et max (inclus)	aucun
@AssertFalse	propriété	vérifie que la méthode est évaluée à faux (utile pour les contraintes exprimées dans le code plutôt que dans les annotations)	aucun
@AssertTrue	propriété	vérifie que la méthode est évaluée à vrai (utile pour les contraintes exprimées dans le code plutôt que dans les annotations)	aucun
@Valid	propriété (objet)	exécute la validation récursivement sur l'objet associé. Si l'objet est une Collection ou un tableau, les éléments sont validés récursivement. Si l'objet est une Map, les éléments valeur sont validés récursivement.	aucun
@Email	propriété (String)	vérifie si la chaîne de caractères est conforme à la spécification d'une adresse e-mail	aucun

4.1.3. Messages d'erreur

Hibernate Validator arrive avec un ensemble de messages d'erreur par défaut traduits dans environ dix langues (si la vôtre n'en fait pas partie, veuillez nous envoyer un patch). Vous pouvez surcharger ces messages en créant un `ValidatorMessages.properties` (ou `ValidatorMessages_loc.properties`) et en surchargeant les clefs dont vous avez besoin. Vous pouvez même ajouter votre propre ensemble de messages supplémentaire lorsque vous écrivez vos annotations de validation. Si Hibernate Validator ne peut pas trouver une clef à partir de votre resourceBundle ou de votre ValidatorMessage, il se repliera sur les valeurs intégrées par défaut.

Alternativement vous pouvez fournir un `ResourceBundle` pendant la vérification par programmation des règles de validation sur un bean, ou si vous voulez un mécanisme d'interpolation complètement différent, vous pouvez fournir une implémentation de `org.hibernate.validator.MessageInterpolator` (lisez la JavaDoc pour plus d'informations).

4.1.4. Ecrire vos propres contraintes

Etendre l'ensemble de contraintes intégrées est extrêment facile. N'importe quelle contrainte est constituée deux morceaux : le *descripteur* de contrainte (l'annotation) et le *validateur* de contrainte (la classe d'implémentation). Voici un simple descripteur personnalisé :

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "has incorrect capitalization";
}
```

`type` est un paramètre décrivant comment la propriété devrait être mise en majuscule. Ceci est un paramètre utilisateur complètement dépendant du fonctionnement de l'annotation.

`message` est la chaîne de caractères par défaut utilisée pour décrire la violation de contrainte et est obligatoire. Vous pouvez mettre la chaîne de caractères dans le code ou bien l'externaliser en partie ou complètement avec le mécanisme ResourceBundle Java. Les valeurs des paramètres sont injectées à l'intérieur du message quand la chaîne de caractères `{parameter}` est trouvée (dans notre exemple `Capitalization is not {type}` générerait `Capitalization is not FIRST`), externaliser toute la chaîne dans `ValidatorMessages.properties` est considéré comme une bonne pratique. Voir Messages d'erreur.

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "{validator.capitalized}";
}

...
#in ValidatorMessages.properties
validator.capitalized=Capitalization is not {type}
```

Comme vous pouvez le voir la notation {} est récursive.

Pour lier un descripteur à l'implémentation de son validateur, nous utilisons la méta-annotation `@ValidatorClass`. Le paramètre de la classe du validateur doit nommer une classe qui implémente `Validator<ConstraintAnnotation>`.

Nous devons maintenant implémenter le validateur (ie l'implémentation vérifiant la règle). Une implémentation de validation peut vérifier la valeur d'une propriété (en implémentant `PropertyConstraint`) et/ou peut modifier les méta-données de mapping d'Hibernate pour exprimer la contrainte au niveau de la base de données (en implémentant `PersistentClassConstraint`).

```
public class CapitalizedValidator
        implements Validator<Capitalized>, PropertyConstraint {
    private CapitalizeType type;

    // partie du contrat de Validator<Annotation>,
    // permet d'obtenir et d'utiliser les valeurs de l'annotation
    public void initialize(Capitalized parameters) {
        type = parameters.type();
    }

    // partie du contrat de la contrainte de la propriété
    public boolean isValid(Object value) {
        if (value==null) return true;
        if ( !(value instanceof String) ) return false;
        String string = (String) value;
        if (type == CapitalizeType.ALL) {
            return string.equals( string.toUpperCase() );
        }
        else {
            String first = string.substring(0,1);
            return first.equals( first.toUpperCase();
        }
    }
}
```

La méthode `isValid()` devrait retourner false si la contrainte a été violée. Pour plus d'exemples, référez-vous aux implémentations intégrées du validateur.

Nous avons seulement vu la validation au niveau propriété, mais vous pouvez écrire une annotation de validation au niveau d'un bean. Plutôt que de recevoir l'instance de retour d'une propriété, le bean lui-même sera passé au validateur. Pour activer la vérification de validation, annotez juste le bean lui-même. Un petit exemple peut être trouvé dans la suite de tests unitaires.

4.1.5. Annoter votre modèle de données

Maintenant que vous vous êtes familiarisés avec les annotations, la syntaxe devrait être connue.

```
public class Address {
    private String line1;
    private String line2;
    private String zip;
    private String state;
    private String country;
    private long id;

    // une chaîne non nulle de 20 caractères maximum
    @Length(max=20)
    @NotNull
    public String getCountry() {
        return country;
    }

    // une chaîne de caractères non nulle
    @NotNull
    public String getLine1() {
        return line1;
    }

    // pas de contrainte
```

```
    public String getLine2() {
        return line2;
    }

    // une chaîne non nulle de 3 caractères maximum
    @Length(max=3) @NotNull
    public String getState() {
        return state;
    }

    // une chaîne non nulle de 5 caractères maximum représentant un nombre
    // si la chaîne de caractères est plus longue, le message sera recherché
    // dans le resource bundle avec la clef 'long'
    @Length(max=5, message="{long}")
    @Pattern(regex="[0-9]+")
    @NotNull
    public String getZip() {
        return zip;
    }

    // devrait toujours être vrai
    @AssertTrue
    public boolean isValid() {
        return true;
    }

    // un nombre entre 1 et 2000
    @Id @Min(1)
    @Range(max=2000)
    public long getId() {
        return id;
    }
}
```

Bien que l'exemple montre seulement la validation de propriétés publiques, vous pouvez aussi annoter des champs avec n'importe quelle visibilité.

```
@MyBeanConstraint(max=45)
public class Dog {
    @AssertTrue private boolean isMale;
    @NotNull protected String getName() { ... };
    ...
}
```

Vous pouvez aussi annoter des inferfaces. Hibernate Validator vérifiera toutes les classes parentes et les interfaces héritées ou implémentées par un bean donné pour lire les annotations appropriées du validateur.

```
public interface Named {
    @NotNull String getName();
    ...
}

public class Dog implements Named {

    @AssertTrue private boolean isMale;

    public String getName() { ... };

}
```

La propriété "name" sera vérifiée pour la nullité lorsque le bean Dog sera validé.

4.2. Utiliser le framework Validator

Hibernate Validator est destiné à être utilisé pour implémenter une validation de données à plusieurs couches, où nous exprimons des contraintes à un seul endroit (le modèle de données annoté) et les appliquons aux différents niveaux de l'application.

4.2.1. Validation au niveau du schéma de la base de données

Par défaut, Hibernate Annotations traduira les contraintes que vous avez définies sur vos entités en méta-données de mapping. Par exemple, si une propriété de votre entité est annotée avec `@NotNull`, ses colonnes seront déclarées comme `not null` dans le schéma DDL généré par Hibernate.

4.2.2. La validation basée sur les événements Hibernate

Hibernate Validator a deux listeners d'événements Hibernate intégrés. Quand un `PreInsertEvent` ou un `PreUpdateEvent` survient, les listeners vérifieront toutes les contraintes de l'instance de l'entité et lèveront une exception si une contrainte est violée. Fondamentalement, les objets seront vérifiés avant les insertions et avant les mises à jour effectuées par Hibernate. C'est le plus commode et la manière la plus simple d'activer le processus de validation. Sur une violation de contrainte, l'événement lèvera une exception d'exécution `InvalidStateException` (NdT : c'est une RuntimeException) laquelle contient un tableau d'`InvalidValues` décrivant chaque échec.

```
<hibernate-configuration>
    ...
    <event type="pre-update">
        <listener
          class="org.hibernate.validator.event.ValidatePreUpdateEventListener"/>
    </event>
    <event type="pre-insert">
        <listener
          class="org.hibernate.validator.event.ValidatePreInsertEventListener"/>
    </event>
</hibernate-configuration>
```

Note

Lors de l'utilisation d'Hibernate Entity Manager, le framework Validation est activé par défaut. Si les beans ne sont pas annotés avec des annotations de validation, il n'y a pas de coût en terme de performance.

4.2.3. La validation au niveau applicatif

Hibernate Validator peut être utilisé n'importe où dans le code de votre application.

```
ClassValidator personValidator = new ClassValidator( Person.class );
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address);
```

Les deux premières lignes préparent Hibernate Validator pour la vérification de classes. La première s'appuie sur les messages d'erreur intégrés à Hibernate Validator (voir Messages d'erreur), la seconde utilise un resource bundle pour ses messages. Il est considéré comme une bonne pratique d'exécuter ces lignes une fois et de cacher les instances de validateur.

La troisième ligne valide en fait l'instance `Address` et retourne un tableau d'`InvalidValues`. Votre logique applicative sera alors capable de réagir aux échecs.

Vous pouvez aussi vérifier une propriété particulière plutôt que tout le bean. Ceci pourrait être utile lors d'interactions avec l'utilisateur propriété par propriété.

```
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

// récupère seulement les valeurs invalides de la propriété "city"
InvalidValue[] validationMessages = addressValidator.getInvalidValues(address, "city");

// récupère seulement les valeurs potentielles invalides de la propriété "city"
InvalidValue[] validationMessages = addressValidator.getPotentialInvalidValues("city", "Paris")
```

4.2.4. Informations de validation

Comme un transporteur d'informations de validation, Hibernate fournit un tableau d'`InvalidValue`s. Chaque `InvalidValue` a un groupe de méthodes décrivant les problèmes individuels.

`getBeanClass()` récupère le type du bean ayant échoué.

`getBean()` récupère l'instance du bean ayant échoué (s'il y en a, c'est-à-dire pas lors de l'utilisation de `getPotentianInvalidValues()`).

`getValue()` récupère la valeur ayant échouée.

`getMessage()` récupère le message d'erreur internationalisé.

`getRootBean()` récupère l'instance du bean racine ayant généré le problème (utile en conjonction avec `@Valid`), est nulle si getPotentianInvalidValues() est utilisée.

`getPropertyPath()` récupère le chemin (séparé par des points) de la propriété ayant échouée à partir du bean racine.

Chapitre 5. Intégration de Lucene avec Hibernate

Lucene est une bibliothèque de la fondation Apache fournissant un moteur de recherche en Java hautement performant. Hibernate Annotations inclut un ensemble d'annotations qui vous permettent de marquer n'importe quel objet du modèle de données comme indexable et de laisser Hibernate maintenir un index Lucene de toutes les instances persistées via Hibernate.

Hibernate Lucene est un projet en cours et de nouvelles fonctionnalités sont en préparation. Donc attendez-vous à certains changements avec les versions ultérieures.

5.1. Mapper les entités sur l'index

Tout d'abord, nous devons déclarer une classe persistante comme étant indexable. Ceci se fait en annotant la classe avec `@Indexed` :

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

L'attribut `index` indique à Hibernate quel est le nom du répertoire Lucene (en général un répertoire de votre système de fichiers). Si vous souhaitez définir un répertoire de départ pour tous vos index Lucene, vous pouvez utiliser la propriété `hibernate.lucene.default.indexDir` dans votre fichier de configuration.

Les index Lucene contiennent quatre types de champs : *keyword*, *text*, *unstored* et *unindexed*. Hibernate Annotations fournit des annotations pour marquer une propriété d'une entité comme étant d'un des trois premiers types.

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

    @Id
    @Keyword(id=true)
    public Long getId() { return id; }

    @Text(name="Abstract")
    public String getSummary() { return summary; }

    @Lob
    @Unstored
    public String getText() { return text; }

}
```

Ces annotations définissent un index avec trois champs : `id`, `Abstract` et `text`. Notez que par défaut le nom du champ n'a plus de majuscules, en suivant la spécification JavaBean.

Note : vous *devez* spécifier `@Keyword(id=true)` sur la propriété identifiante de votre entité.

Lucene a la notion of *boost factor*. C'est un moyen de donner plus de poids à un champ ou à un élément indexé durant la procédure d'indexation. Vous pouvez utiliser `@Boost` au niveau du champ ou de la classe.

La classe analyste utilisée pour indexer les éléments est configurable par la propriété `hibernate.lucene.analyzer`. Si aucune n'est définie,

`org.apache.lucene.analysis.standard.StandardAnalyzer` est utilisée par défaut.

5.2. Configuration

5.2.1. Configuration du directory

Lucene a une notion de Directory où l'index est stocké. L'implémentation de Directory peut être personnalisée mais Lucene arrive, avec deux implémentations prêtes à l'emploi complètes, une sur un système de fichiers et une en mémoire. Hibernate Lucene a la notion de `DirectoryProvider` qui gère la configuration et l'initialisation du Directory Lucene.

Tableau 5.1. Liste des Directory Providers intégrés

Classe	Description	Propriétés
org.hibernate.lucene.store.FSDirectoryProvider	Directory base sur le système de fichiers. Le répertoire utilisé sera <indexBase>/<`@Index.name`>	`indexBase` : répertoire de départ
org.hibernate.lucene.store.RAMDirectoryProvider	Directory utilisant la mémoire, le directory sera uniquement identifié par l'élément `@Index.name`	aucune

Si les directory providers intégrés ne répondent pas à vos besoins, vous pouvez écrire votre propre directory provider en implémentant l'interface `org.hibernate.store.DirectoryProvider`.

Chaque entité indexée est associée à un index Lucene (un index peut être partagé par différentes entités mais ce n'est pas le cas en général). Vous pouvez configurer l'index à travers des propriétés préfixées par `hibernate.lucene.<indexname>`. Les propriétés par défaut héritées par tous les index peuvent être définies en utilisant le préfixe hibernate.lucene.default.

Pour définir le directory provider d'un index donné, utilisez `hibernate.lucene.<indexname>.directory_provider`.

```
hibernate.lucene.default.directory_provider org.hibernate.lucene.store.FSDirectoryProvider
hibernate.lucene.default.indexDir=/usr/lucene/indexes

hibernate.lucene.Rules.directory_provider org.hibernate.lucene.store.RAMDirectoryProvider
```

appliqué à

```
@Indexed(name="Status")
public class Status { ... }

@Indexed(name="Rules")
public class Rule { ... }
```

Ceci créera un directory système de fichiers dans `/usr/lucene/indexes/Status` où les entités Status seront indexées, et utilisera un directory mémoire nommé `Rules` où les entités Rule seront indexées.

Donc vous pouvez facilement définir des règles générales comme le directory provider et le répertoire de départ, et surcharger ces valeurs par défaut plus tard pour chaque index.

En écrivant votre propre DirectoryProvider, vous pouvez aussi bénéficier de ce mécanisme de configuration.

5.2.2. Activer l'indexation automatique

Finalement, nous activons le `LuceneEventListener` pour les trois événements Hibernate qui ont lieu après que les changements sont validés dans la base de données.

```
<hibernate-configuration>
    ...
    <event type="post-commit-update"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
    <event type="post-commit-insert"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
    <event type="post-commit-delete"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
</hibernate-configuration>
```