Hibernate Annotations

Reference Guide

3.5.6-Final

by Emmanuel Bernard

Preface

Hibernate, like all other object/relational mapping tools, requires metadata that governs the transformation of data from one representation to the other. Hibernate Annotations provides annotation-based mapping metadata.

The JPA specification recognizes the interest and the success of the transparent object/relational mapping paradigm. It standardizes the basic APIs and the metadata needed for any object/relational persistence mechanism. *Hibernate EntityManager* implements the programming interfaces and lifecycle rules as defined by the JPA persistence specification and together with *Hibernate Annotations* offers a complete (and standalone) JPA persistence solution on top of the mature Hibernate Core. You may use a combination of all three together, annotations without JPA programming interfaces and lifecycle, or even pure native Hibernate Core, depending on the business and technical needs of your project. At all time you can fall back to Hibernate native APIs, or if required, even to native JDBC and SQL.

This release of *Hibernate Annotations* is based on the final release of the JPA 2 specification (aka *JSR-317* [http://jcp.org/en/jsr/detail?id=317]) and supports all its features (including the optional ones). Hibernate specific features and extensions are also available through unstandardized, Hibernate specific annotations.

If you are moving from previous Hibernate Annotations versions, please have a look at *Java Persistence migration guide* [http://www.hibernate.org/398.html].

Setting up an annotations project

1.1. Requirements

- Make sure you have JDK 5.0 or above installed.
- Download and unpack the Hibernate Core distribution from the Hibernate website. Hibernate 3.5 and onward contains Hibernate Annotations.
- Alternatively add the following dependency in your dependency manager (like Maven or Ivy). Here is an example

```
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate-core-version}</version>
    </dependency>
  </dependencies>
</project>
```

1.2. Configuration

First, set up your classpath (after you have created a new project in your favorite IDE):

- Copy `hibernate3.jar` and the required 3rd party libraries available in `lib/required`.
- Copy `lib/jpa/hibernate-jpa-2.0-api-1.0.0.Final.jar` to your classpath as well.

Alternatively, import your pom.xml in your favorite IDE and let the dependencies be resolved automatically,

What is hibernate-jpa-2.0-api-x.y.z.jar

This is the JAR containing the JPA 2.0 API, it is fully compliant with the spec and passed the TCK signature test. You typically don't need it when you deploy your application in a Java EE 6 application server (like JBoss AS 6 for example).

We recommend you use *Hibernate Validator* [http://validator.hibernate.org] and the Bean Validation specification capabilities as its integration with Java Persistence 2 has been standardized. Download Hibernate Validator 4 or above from the Hibernate website and add `hibernate-validator.jar` and `validation-api.jar` in your classpath. Alternatively add the following dependency in your `pom.xml`.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>${hibernate-validator-version}</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

If you wish to use *Hibernate Search* [http://search.hibernate.org], download it from the Hibernate website and add `hibernate-search.jar` and its dependencies in your classpath. Alternatively add the following dependency in your `pom.xml`.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search</artifactId>
      <version>${hibernate-search-version}</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

We recommend you use the JPA 2 APIs to bootstrap Hibernate (see the Hibernate EntityManager documentation for more information). If you use Hibernate Core and its native APIs read on.

If you boot Hibernate yourself, make sure to use the `AnnotationConfiguration` class instead of the `Configuration` class. Here is an example using the (legacy) `HibernateUtil` approach:

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

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

Interesting here is the use of `AnnotationConfiguration`. The packages and annotated classes are declared in your regular XML configuration file (usually `hibernate.cfg.xml`). Here is the equivalent of the above declaration:

```
<!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>
```

Note that you can mix the legacy hbm.xml use and the annotation approach. The resource element can be either an hbm file or an EJB3 XML deployment descriptor. The distinction is transparent for your configuration process.

Alternatively, you can define the annotated classes and packages using the programmatic API

```
sessionFactory = new AnnotationConfiguration()
                    .addPackage("test.animals") //the fully qualified package name
                    .addAnnotatedClass(Flight.class)
                    .addAnnotatedClass(Sky.class)
                    .addAnnotatedClass(Person.class)
                    .addAnnotatedClass(Dog.class)
                    .addResource("test/animals/orm.xml")
                    .configure()
                    .buildSessionFactory();
```

There is no other difference in the way you use Hibernate APIs with annotations, except for this startup routine change or in the configuration file. You can use your favorite configuration method for other properties (`hibernate.properties`, `hibernate.cfg.xml`, programmatic APIs, etc).

Note

You can mix annotated persistent classes and classic `hbm.cfg.xml` declarations with the same `SessionFactory`. You can however not declare a class several times (whether annotated or through hbm.xml). You cannot mix configuration strategies (hbm vs annotations) in an entity hierarchy either.

To ease the migration process from hbm files to annotations, the configuration mechanism detects the mapping duplication between annotations and hbm files. HBM files are then prioritized over annotated metadata on a class to class basis. You can change the priority using `hibernate.mapping.precedence` property. The default is `hbm, class`, changing it to `class, hbm` will prioritize the annotated classes over hbm files when a conflict occurs.

1.3. Properties

On top of the Hibernate Core properties, Hibernate Annotations reacts to the following one.

Table 1.1. Hibernate Annotations specific properties

Property	Function
`hibernate.cache.default_cache_concurrency_strategy`	Setting used to give the name of the default `org.hibernate.annotations.CacheConcurrencyStrategy` to use when either `@Cacheable @Cache`} is used. `@Cache(strategy="..")` is used to override this default.
`hibernate.id.new_generator_mappings`	true or false. Setting which indicates whether or not the new `IdentifierGenerator` implementations are used for AUTO, TABLE and SEQUENCE. Default to false to keep backward compatibility.

Note

We recommend all new projects to use `hibernate.id.new_generator_mappings=true` as the new generators are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

1.4. Logging

Hibernate Annotations utilizes *Simple Logging Facade for Java* [http://www.slf4j.org/] (SLF4J) in order to log various system events. SLF4J can direct your logging output to several logging frameworks (NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL or logback) depending on your chosen binding. In order to setup logging properly you will need `slf4j-api.jar` in your classpath together with the jar file for your preferred binding - `slf4j-log4j12.jar` in the case of Log4J. See the SLF4J *documentation* [http://www.slf4j.org/manual.html] for more detail.

The logging categories interesting for Hibernate Annotations are:

Table 1.2. Hibernate Annotations Log Categories

Category	Function
org.hibernate.cfg	Log all configuration related events (not only annotations).

For further category configuration refer to the *Logging* [http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#configuration-logging] in the Hibernate Core documentation.

Mapping Entities

2.1. Intro

This section explains how to describe persistence mappings using Java Persistence 2.0 annotations as well as Hibernate-specific annotation extensions.

2.2. Mapping with JPA (Java Persistence Annotations)

JPA entities are plain POJOs. Actually, they are Hibernate persistent entities. Their mappings are defined through JDK 5.0 annotations instead of hbm.xml files. A JPA 2 XML descriptor syntax for overriding is defined as well). Annotations can be split in two categories, the logical mapping annotations (describing the object model, the association between two entities etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

JPA annotations are in the `javax.persistence.*` package. You favorite IDE can auto-complete annotations and their attributes for you (even without a specific "JPA" module, since JPA annotations are plain JDK 5 annotations).

A good an complete set of working examples can be found in the Hibernate Annotations test suite itself: most of the unit tests have been designed to represent a concrete example and be a source of inspiration for you. You can get the test suite sources in the distribution.

2.2.1. Marking a POJO as persistent entity

Every persistent POJO class is an entity and is declared using the `@Entity` annotation (at the class level):

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

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

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

`@Entity` declares the class as an entity (i.e. a persistent POJO class), `@Id` declares the identifier property of this entity. The other mapping declarations are implicit. The class Flight is mapped to the Flight table, using the column id as its primary key column.

Note

The concept of configuration by exception is central to the JPA specification.

Depending on whether you annotate fields or methods, the access type used by Hibernate will be `field` or `property`. The EJB3 spec requires that you declare annotations on the element type that will be accessed, i.e. the getter method if you use `property` access, the field if you use `field` access. Mixing annotations in both fields and methods should be avoided. Hibernate will guess the access type from the position of `@Id` or `@EmbeddedId`.

2.2.1.1. Defining the table

`@Table` is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping. If no `@Table` is defined the default values are used: the unqualified class name of the entity.

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

The `@Table` element contains a `schema` and `catalog` attributes, if they need to be defined. You can also define unique constraints to the table using the `@UniqueConstraint` annotation in conjunction with `@Table` (for a unique constraint bound to a single column, it is recommended to use the `@Column.unique` approach (refer to `@Column` for more information).

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

A unique constraint is applied to the tuple month, day. Note that the `columnNames` array refers to the logical column names.
The logical column name is defined by the Hibernate `NamingStrategy` implementation. The default JPA naming strategy uses the physical column name as the logical column name but it could be different if for example you append fld_ to all your columns using a custom `NamingStrategy` implementation. Note that the logical column name is not necessarily equals to the property name esp when the column name is explicitly set. Unless you override the `NamingStrategy`, you shouldn't worry about that.

2.2.1.2. Versioning for optimistic locking

You can add optimistic locking capability to an entity using the `@Version` annotation:

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

The version property will be mapped to the `OPTLOCK` column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric (the recommended solution) or a timestamp. Hibernate supports any kind of type provided that you define and implement the appropriate `UserVersionType`.

The application must not alter the version number set up by Hibernate in any way. To artificially increase the version number, check in Hibernate Entity Manager's reference documentation `LockModeType.OPTIMISTIC_FORCE_INCREMENT` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT`.

2.2.2. Mapping simple properties

2.2.2.1. Declaring basic property mappings

Every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as `@Transient`. Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation. The `@Basic` annotation allows you to declare the fetching strategy for a property:

```
public transient int counter; //transient property

private String firstname; //persistent property

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

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

`counter`, a transient field, and `lengthInMeter`, a method annotated as `@Transient`, and will be ignored by the entity manager. `name`, `length`, and `firstname` properties are mapped persistent and eagerly fetched (the default for simple properties). The `detailedComment` property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching).

Note

To enable property level lazy fetching, your classes have to be instrumented: bytecode is added to the original class to enable such feature, please refer to the Hibernate reference documentation. If your classes are not instrumented, property level lazy loading is silently ignored.

The recommended alternative is to use the projection capability of JP-QL (Java Persistence Query Language) or Criteria queries.

JPA support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations support out of the box enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overridden through the `@Enumerated` annotation as shown in the `note` property example.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have `DATE`, `TIME`, or `TIMESTAMP` precision (ie the actual date, only the time, or both). Use the `@Temporal` annotation to fine tune that.

`@Lob` indicates that the property should be persisted in a Blob or a Clob depending on the property type: `java.sql.Clob`, `Character[]`, `char[]` and java.lang.`String` will be persisted in a Clob. `java.sql.Blob`, `Byte[]`, `byte[]` and serializable type will be persisted in a Blob.

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

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

If the property type implements `java.io.Serializable` and is not a basic type, and if the property is not annotated with `@Lob`, then the Hibernate `serializable` type is used.

2.2.2.2. Access type

By default the access type of a class hierarchy is defined by the position of the `@Id` or `@EmbeddedId` annotations. If these annotations are on a field, then only fields are considered for persistence and the state is accessed via the field. If there annotations are on a getter, then only the getters are considered for persistence and the state is accessed via the getter/setter. That works well in practice and is the recommended approach.

Note

The placement of annotations within a class hierarchy has to be consistent (either field or on property) to be able to determine the default access type. It is recommended to stick to one single annotation placement strategy throughout your whole application.

However in some situations, you need to:

- force the access type of the entity hierarchy
- override the access type of a specific entity in the class hierarchy
- override the access type of an embeddable type

The best use case is an embeddable class used by several entities that might not use the same access type. In this case it is better to force the access type at the embeddable class level.

To force the access type on a given class, use the `@Access` annotation as showed below:

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   @Embedded private Address address;
   public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Entity
public class User {
   private Long id;
   @Id public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   private Address address;
   @Embedded public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Embeddable
```

```
@Access(AcessType.PROPERTY)
public class Address {
   private String street1;
   public String getStreet1() { return street1; }
   public void setStreet1() { this.street1 = street1; }

   private hashCode; //not persistent
}
```

You can also override the access type of a single property while keeping the other properties standard.

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   @Transient private String userId;
   @Transient private String orderId;

   @Access(AccessType.PROPERTY)
   public String getOrderNumber() { return userId + ":" + orderId; }
   public void setOrderNumber() { this.userId = ...; this.orderId = ...; }
}
```

In this example, the default access type is `FIELD` except for the `orderNumber` property. Note that the corresponding field, if any must be marked as `@Transient` or `transient`.

@org.hibernate.annotations.AccessType

The annotation `@org.hibernate.annotations.AccessType` should be considered deprecated for FIELD and PROPERTY access. It is still useful however if you need to use a custom access type.

2.2.2.3. Declaring column attributes

The column(s) used for a property mapping can be defined using the `@Column` annotation. Use it to override default values (see the EJB3 specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with `@Basic`
- annotated with `@Version`
- annotated with `@Lob`
- annotated with `@Temporal`

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

The `name` property is mapped to the `flight_name` column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as `@Id` or `@Version` properties.

```
@Column(
    name="columnName";                                                    ❶
    boolean unique() default false;                                       ❷
    boolean nullable() default true;                                      ❸
    boolean insertable() default true;                                    ❹
    boolean updatable() default true;                                     ❺
    String columnDefinition() default "";                                 ❻
    String table() default "";                                            ❼
    int length() default 255;                                             ❽
    int precision() default 0; // decimal precision                       ❾
    int scale() default 0; // decimal scale
```

❶ `name` (optional): the column name (default to the property name)

❷ `unique` (optional): set a unique constraint on this column or not (default false)

❸ `nullable` (optional): set the column as nullable (default true).

❹ `insertable` (optional): whether or not the column will be part of the insert statement (default true)

❺ `updatable` (optional): whether or not the column will be part of the update statement (default true)

❻ `columnDefinition` (optional): override the sql DDL fragment for this particular column (non portable)

❼ `table` (optional): define the targeted table (default primary table)

❽ `length` (optional): column length (default 255)

❽ `precision` (optional): column decimal precision (default 0)

❿ `scale` (optional): column decimal scale if useful (default 0)

2.2.2.4. Embedded objects (aka components)

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the `@Embeddable` annotation. It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and `@AttributeOverride` annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    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; //no overriding here
}
```

```
@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; }
    ...
}
```

An embeddable object inherits the access type of its owning entity (note that you can override that using the `@Access` annotation).

The `Person` entity has two component properties, `homeAddress` and `bornIn`. `homeAddress` property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the `@Embeddable` annotation in the Address class. We also override the mapping of a column name (to `bornCountryName`) with the `@Embedded` and `@AttributeOverride` annotations for each mapped attribute of `Country`. As you can see, `Country` is also a nested component of `Address`, again using auto-detection by Hibernate and JPA defaults. Overriding columns of embedded objects of embedded objects is through dotted expressions.

```
@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") )
        //nationality columns in homeAddress are overridden
} )
Address homeAddress;
```

Hibernate Annotations supports something that is not explicitly supported by the JPA specification. You can annotate a embedded object with the `@MappedSuperclass` annotation to make the superclass properties persistent (see `@MappedSuperclass` for more informations).

You can also use association annotations in an embeddable object (ie `@OneToOne`, `@ManyToOne`, `@OneToMany` or `@ManyToMany`). To override the association columns you can use `@AssociationOverride`.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work as several embedded objects would share the same set of columns. In plain JPA, you need to override at least one set of columns. Hibernate, however, allows you to enhance the default naming mechanism through the `NamingStrategy` interface. You can write a strategy that prevent name clashing in such a situation. `DefaultComponentSafeNamingStrategy` is an example of this.

2.2.2.5. Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is `Serializable`, it is mapped as `@Basic` in a column holding the object in its serialized version
- Otherwise, if the type of the property is `java.sql.Clob` or `java.sql.Blob`, it is mapped as `@Lob` with the appropriate `LobType`

2.2.3. Mapping identifier properties

The `@Id` annotation lets you define which property is the identifier of your entity. This property can be set by the application itself or be generated by Hibernate (preferred). You can define the identifier generation strategy thanks to the `@GeneratedValue` annotation.

2.2.3.1. Generating the identifier property

JPA defines five types of identifier generation strategies:

- AUTO - either identity column, sequence or table depending on the underlying DB
- TABLE - table holding the id
- IDENTITY - identity column
- SEQUENCE - sequence
- identity copy - the identity is copied from another entity

Hibernate provides more id generators than the basic JPA ones. Check *Section 2.4, “Hibernate Annotation Extensions”* for more informations.

The following example shows a sequence generator using the SEQ_STORE configuration (see below)

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

The next example uses the identity generator:

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

The `AUTO` generator is the preferred type for portable applications (across several DB vendors). The identifier generation configuration can be shared for several `@Id` mappings with the generator attribute. There are several configurations available through `@SequenceGenerator` and `@TableGenerator`. The scope of a generator can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined at XML level (see *Chapter 3, Overriding metadata through XML*):

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

//and the annotation equivalent

@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"/>

//and the annotation equivalent

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

If JPA XML (like `META-INF/orm.xml`) is used to define the generators, `EMP_GEN` and `SEQ_GEN` are application level generators. `EMP_GEN` defines a table based id generator using the hilo algorithm with a `max_lo` of 20. The hi value is kept in a `table` "`GENERATOR_TABLE`". The information is kept in a row where `pkColumnName` "key" is equals to `pkColumnValue` "`EMP`" and column `valueColumnName` "`hi`" contains the the next high value used.

`SEQ_GEN` defines a sequence generator using a sequence named `my_sequence`. The allocation size used for this sequence based hilo algorithm is 20. Note that this version of Hibernate Annotations does not handle `initialValue` in the sequence generator. The default allocation size is 50, so if you want to use a sequence and pickup the value each time, you must set the allocation size to 1.

Important

We recommend all new projects to use `hibernate.id.new_generator_mappings=true` as the new generators are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation). See *Section 1.3, “Properties”* for more information on how to activate them.

Note

Package level definition is not supported by the JPA specification. However, you can use the `@GenericGenerator` at the package level (see *Section 2.4.2, “Identifier”*).

The next example shows the definition of a sequence generator in a class scope:

```
@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; }
}
```

This class will use a sequence named my_sequence and the SEQ_STORE generator is not visible in other classes. Note that you can check the Hibernate Annotations tests in the org.hibernate.test.annotations.id package for more examples.

Finally, you can ask Hibernate to copy the identifier from another associated entity. In the Hibernate jargon, it is known as a foreign generator but the JPA mapping reads better and is encouraged.

```
@Entity
class MedicalHistory implements Serializable {
  @Id @OneToOne
  @JoinColumn(name = "person_id")
  Person patient;
}

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

Or alternatively

```
@Entity
class MedicalHistory implements Serializable {
  @Id Integer id;

  @MapsId @OneToOne
  @JoinColumn(name = "patient_id")
  Person patient;
}

@Entity
class Person {
  @Id @GeneratedValue Integer id;
}
```

If you are interested in more examples of "derived identities", the JPA 2 specification has a great set of them in chapter 2.4.1.3.

But an identifier does not have to be a single property, it can be composed of several properties.

2.2.3.2. Composite identifier

You can define a composite primary key through several syntaxes:

- use a component type to represent the identifier and map it as a property in the entity: you then annotated the property as `@EmbeddedId`. The component type has to be `Serializable`.
- map multiple properties as `@Id` properties: the identifier type is then the entity class itself and needs to be `Serializable`. This approach is unfortunately not standard and only supported by Hibernate.
- map multiple properties as `@Id` properties and declare an external class to be the identifier type. This class, which needs to be `Serializable`, is declared on the entity via the `@IdClass` annotation. The identifier type must contain the same properties as the identifier properties of the entity: each property name must be the same, its type must be the same as well if the entity property is of a basic type, its type must be the type of the primary key of the associated entity if the entity property is an association (either a `@OneToOne` or a `@ManyToOne`).

As you can see the last case is far from obvious. It has been inherited from the dark ages of EJB 2 for backward compatibilities and we recommend you not to use it (for simplicity sake).

Let's explore all three cases using examples.

2.2.3.2.1. @EmbeddedId property

Here is a simple example of `@EmbeddedId`.

```
@Entity
class User {
  @EmbeddedId
  @AttributeOverride(name="firstName", column=@Column(name="fld_firstname")
  UserId id;

  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

You can notice that the `UserId` class is serializable. To override the column mapping, use `@AttributeOverride`.

An embedded id can itself contains the primary key of an associated entity.

```
@Entity
class Customer {
  @EmbeddedId CustomerId id;
  boolean preferredCustomer;

  @MapsId("userId")
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
  })
  @OneToOne User user;
}

@Embeddable
class CustomerId implements Serializable {
  UserId userId;
  String customerNumber;
}

@Entity
class User {
  @EmbeddedId UserId id;
  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

In the embedded id object, the association is represented as the identifier of the associated entity. But you can link its value to a regular association in the entity via the `@MapsId` annotation. The `@MapsId` value correspond to the property name of the embedded id object containing the associated entity's identifier. In the database, it means that the `Customer.user` and the `CustomerId.userId` properties share the same underlying column (`user_fk` in this case).

In practice, your code only sets the `Customer.user` property and the user id value is copied by Hibernate into the `CustomerId.userId` property.

Warning

The id value can be copied as late as flush time, don't rely on it until after flush time.

While not supported in JPA, Hibernate lets you place your association directly in the embedded id component (instead of having to use the `@MapsId` annotation).

```
@Entity
class Customer {
  @EmbeddedId CustomerId id;
  boolean preferredCustomer;
}

@Embeddable
```

```
class CustomerId implements Serializable {
  @OneToOne
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
  })
  User user;
  String customerNumber;
}

@Entity
class User {
  @EmbeddedId UserId id;
  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

2.2.3.2.2. Multiple @Id properties

Another, arguably more natural, approach is to place `@Id` on multiple properties of my entity. This approach is only supported by Hibernate but does not require an extra embeddable component.

```
@Entity
class Customer implements Serializable {
  @Id @OneToOne
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
  })
  User user;

  @Id String customerNumber;

  boolean preferredCustomer;
}

@Entity
class User {
  @EmbeddedId UserId id;
  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

In this case `Customer` being it's own identifier representation, it must implement `Serializable`.

2.2.3.2.3. @IdClass

`@IdClass` on an entity points to the class (component) representing the identifier of the class. The properties marked `@Id` on the entity must have their corresponding property on the `@IdClass`. The return type of search twin property must be either identical for basic properties or must correspond to the identifier class of the associated entity for an association.

Warning

This approach is inherited from the EJB 2 days and we recommend against its use. But, after all it's your application and Hibernate supports it.

```
@Entity
class Customer {
  @Id @OneToOne
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
  })
  User user;

  @Id String customerNumber;

  boolean preferredCustomer;
}

class CustomerId implements Serializable {
  UserId user;
  String customerNumber;
}

@Entity
class User {
  @EmbeddedId UserId id;
  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

`Customer` and `CustomerId` do have the same properties `customerNumber` as well as `user`.

While not JPA standard, Hibernate let's you declare the vanilla associated property in the `@IdClass`.

```
@Entity
class Customer {
  @Id @OneToOne
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
  })
  User user;

  @Id String customerNumber;

  boolean preferredCustomer;
}

class CustomerId implements Serializable {
  @OneToOne User user;
  String customerNumber;
}

@Entity
class User {
  @EmbeddedId UserId id;
  Integer age;
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

2.2.3.2.4. Partial identifier generation

Hibernate supports the automatic generation of some of the identifier properties. Simply use the `@GeneratedValue` annotation on one or several id properties.

Warning

The Hibernate team has always felt such a construct as fundamentally wrong. Try hard to fix your data model before using this feature.

```
@Entity
public class CustomerInventory implements Serializable {
  @Id
  @TableGenerator(name = "inventory",
    table = "U_SEQUENCES",
    pkColumnName = "S_ID",
    valueColumnName = "S_NEXTNUM",
    pkColumnValue = "inventory",
    allocationSize = 1000)
  @GeneratedValue(strategy = GenerationType.TABLE, generator = "inventory")
  Integer id;


  @Id @ManyToOne(cascade = CascadeType.MERGE)
  Customer customer;
}

@Entity
public class Customer implements Serializable {
   @Id
   private int id;
}
```

You can also generate properties inside an `@EmbeddedId` class.

2.2.4. Mapping inheritance

EJB3 supports the three types of inheritance:

- Table per Class Strategy: the <union-class> element in Hibernate
- Single Table per Class Hierarchy Strategy: the <subclass> element in Hibernate
- Joined Subclass Strategy: the <joined-subclass> element in Hibernate

The chosen strategy is declared at the class level of the top level entity in the hierarchy using the `@Inheritance` annotation.

Note

Annotating interfaces is currently not supported.

2.2.4.1. Table per class

This strategy has many drawbacks (esp. with polymorphic queries and associations) explained in the JPA spec, the Hibernate reference documentation, Hibernate in Action, and many other places. Hibernate work around most of them implementing this strategy using `SQL UNION` queries. It is commonly used for the top level of an inheritance hierarchy:

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

This strategy supports one-to-many associations provided that they are bidirectional. This strategy does not support the `IDENTITY` generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use `AUTO` nor `IDENTITY`.

2.2.4.2. Single table per class hierarchy

All properties of all super- and subclasses are mapped into the same table, instances are distinguished by a special discriminator column:

```
@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` is the superclass, it defines the inheritance strategy `InheritanceType.SINGLE_TABLE`. It also defines the discriminator column through the `@DiscriminatorColumn` annotation, a discriminator column can also define the discriminator type. Finally, the `@DiscriminatorValue` annotation defines the value used to differentiate a class in the hierarchy. All of these attributes have sensible default values. The default name of the discriminator column is `DTYPE`. The default discriminator value is the entity name (as defined in `@Entity.name`) for DiscriminatorType.STRING. `A320` is a subclass; you only have to define discriminator value if you don't want to use the default value. The strategy and the discriminator type are implicit.

`@Inheritance` and `@DiscriminatorColumn` should only be defined at the top of the entity hierarchy.

2.2.4.3. Joined subclasses

The `@PrimaryKeyJoinColumn` and `@PrimaryKeyJoinColumns` annotations define the primary key(s) of the joined subclass table:

```
@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 { ... }
```

All of the above entities use the `JOINED` strategy, the `Ferry` table is joined with the `Boat` table using the same primary key names. The `AmericaCupClass` table is joined with `Boat` using the join condition `Boat.id = AmericaCupClass.BOAT_ID`.

2.2.4.4. Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as `@MappedSuperclass`.

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

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

In database, this hierarchy will be represented as an `Order` table having the `id`, `lastUpdate` and `lastUpdater` columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

Note

Properties from superclasses not mapped as `@MappedSuperclass` are ignored.

Note

The default access type (field or methods) is used, unless you use the `@Access` annotation.

Note

The same notion can be applied to `@Embeddable` objects to persist properties from their superclasses. You also need to use `@MappedSuperclass` to do that (this should not be considered as a standard EJB3 feature though)

Note

It is allowed to mark a class as `@MappedSuperclass` in the middle of the mapped inheritance hierarchy.

Note

Any class in the hierarchy non annotated with `@MappedSuperclass` nor `@Entity` will be ignored.

You can override columns defined in entity superclasses at the root entity level using the `@AttributeOverride` annotation.

```
@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 {
    ...
}
```

The `altitude` property will be persisted in an `fld_altitude` column of table `Plane` and the propulsion association will be materialized in a `fld_propulsion_fk` foreign key column.

You can define `@AttributeOverride`(s) and `@AssociationOverride`(s) on `@Entity` classes, `@MappedSuperclass` classes and properties pointing to an `@Embeddable` object.

2.2.5. Mapping entity associations/relationships

2.2.5.1. One-to-one

You can associate entities through a one-to-one relationship using `@OneToOne`. There are three cases for one-to-one associations: either the associated entities share the same primary keys values, a foreign key is held by one of the entities (note that this FK column in the database should be constrained unique to simulate one-to-one multiplicity), or a association table is used to store the link between the 2 entities (a unique constraint has to be defined on each fk to ensure the one to one multiplicity).

First, we map a real one-to-one association using shared primary keys:

```
@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() { ...}
}
```

The `@PrimaryKeyJoinColumn` annotation does say that the primary key of the entity is used as the foreign key value to the associated entity.

In the following example, the associated entities are linked through an explicit foreign key column:

```
@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() {
```

```
    ...
}
```

A `Customer` is linked to a `Passport`, with a foreign key column named `passport_fk` in the `Customer` table. The join column is declared with the `@JoinColumn` annotation which looks like the `@Column` annotation. It has one more parameters named `referencedColumnName`. This parameter declares the column in the targeted entity that will be used to the join. Note that when using `referencedColumnName` to a non primary key column, the associated class has to be `Serializable`. Also note that the `referencedColumnName` to a non primary key column has to be mapped to a property having a single column (other cases might not work).

The association may be bidirectional. In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. To declare a side as *not* responsible for the relationship, the attribute `mappedBy` is used. `mappedBy` refers to the property name of the association on the owner side. In our case, this is `passport`. As you can see, you don't have to (must not) declare the join column since it has already been declared on the owners side.

If no `@JoinColumn` is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column(s) in the owned side. In this example `passport_id` because the property name is `passport` and the column id of `Passport` is `id`.

The third possibility (using an association table) is quite exotic.

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

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

A `Customer` is linked to a `Passport` through a association table named `CustomerPassports` ; this association table has a foreign key column named `passport_fk` pointing to the `Passport` table (materialized by the `inverseJoinColumn`, and a foreign key column named `customer_fk` pointing to the `Customer` table materialized by the `joinColumns` attribute.

You must declare the join table name and the join columns explicitly in such a mapping.

2.2.5.2. Many-to-one

Many-to-one associations are declared at the property level with the annotation `@ManyToOne`:

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

The `@JoinColumn` attribute is optional, the default value(s) is like in one to one, the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column in the owned side. In this example `company_id` because the property name is `company` and the column id of Company is `id`.

`@ManyToOne` has a parameter named `targetEntity` which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

```
@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 {
    ...
}
```

You can also map a many-to-one association through an association table. This association table described by the `@JoinTable` annotation will contains a foreign key referencing back the entity table (through `@JoinTable.joinColumns`) and a a foreign key referencing the target entity table (through `@JoinTable.inverseJoinColumns`).

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

2.2.5.3. Collections

You can map `Collection`, `List`, `Map` and `Set` pointing to associated entities as one-to-many or many-to-many associations using the `@OneToMany` or `@ManyToMany` annotation respectively. If the collection is of a basic type or of an embeddable type, use `@ElementCollection`. We will describe that in more detail in the following subsections.

2.2.5.3.1. One-to-many

One-to-many associations are declared at the property level with the annotation `@OneToMany`. One to many associations may be bidirectional.

2.2.5.3.1.1. Bidirectional

Since many to one are (almost) always the owner side of a bidirectional relationship in the JPA spec, the one to many association is annotated by `@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` has a bidirectional one to many relationship with `Soldier` through the `troop` property. You don't have to (must not) define any physical mapping in the `mappedBy` side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the `mappedBy` element and set the many to one `@JoinColumn` as insertable and updatable to false. This solution is not optimized and will produce some additional UPDATE statements.

```
@Entity
```

```
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

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

2.2.5.3.1.2. Unidirectional

A unidirectional one to many using a foreign key column in the owned entity is not that common and not really recommended. We strongly advise you to use a join table for this kind of association (as explained in the next section). This kind of association is described through a `@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 {
    ... //no bidir
}
```

`Customer` describes a unidirectional relationship with `Ticket` using the join column `CUST_ID`.

2.2.5.3.1.3. Unidirectional with join table

A unidirectional one to many with join table is much preferred. This association is described through an `@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 {
    ... //no bidir
}
```

`Trainer` describes a unidirectional relationship with `Monkey` using the join table `TrainedMonkeys`, with a foreign key `trainer_id` to `Trainer` (`joinColumns`) and a foreign key `monkey_id` to `Monkey` (`inversejoinColumns`).

2.2.5.3.1.4. Defaults

Without describing any physical mapping, a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

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

@Entity
public class Tiger {
    ... //no bidir
}
```

`Trainer` describes a unidirectional relationship with `Tiger` using the join table `Trainer_Tiger`, with a foreign key `trainer_id` to `Trainer` (table name, _, trainer id) and a foreign key `trainedTigers_id` to `Monkey` (property name, _, Tiger primary column).

2.2.5.3.2. Many-to-many

2.2.5.3.2.1. Definition

A many-to-many association is defined logically using the `@ManyToMany` annotation. You also have to describe the association table and the join conditions using the `@JoinTable` annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

```
@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;
    }
}
```

We've already shown the many declarations and the detailed attributes for associations. We'll go deeper in the `@JoinTable` description, it defines a `name`, an array of join columns (an array in annotation is defined using { A, B, C }), and an array of inverse join columns. The latter ones are the columns of the association table which refer to the `Employee` primary key (the "other side").

As seen previously, the other side don't have to (must not) describe the physical mapping: a simple `mappedBy` argument containing the owner side property name bind the two.

2.2.5.3.2.2. Default values

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

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

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A `Store_City` is used as the join table. The `Store_id` column is a foreign key to the `Store` table. The `implantedIn_id` column is a foreign key to the `City` table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

```
@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() {
        ...
    }
}
```

A `Store_Customer` is used as the join table. The `stores_id` column is a foreign key to the `Store` table. The `customers_id` column is a foreign key to the `Customer` table.

2.2.5.3.3. Collection of basic types or embeddable objects

In some simple situation, do don't need to associate two entities but simply create a collection of basic types or embeddable objects. Use the `@ElementCollection` in this case.

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
   @Column(name="nickname")
   public Set<String> getNicknames() { ... }
}
```

The collection table holding the collection data is set using the `@CollectionTable` annotation. If omitted the collection table name default to the concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore: in our example, it would be `User_nicknames`.

The column holding the basic type is set using the `@Column` annotation. If omitted, the column name defaults to the property name: in our example, it would be `nicknames`.

But you are not limited to basic types, the collection type can be any embeddable object. To override the columns of the embeddable object in the collection table, use the `@AttributeOverride` annotation.

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
   @AttributeOverrides({
      @AttributeOverride(name="street1", column=@Column(name="fld_street"))
   })
   public Set<Address> getAddresses() { ... }
}

@Embeddable
public class Address {
   public String getStreet1() {...}
   [...]
}
```

Such an embeddable object cannot contains a collection itself.

Note

in `@AttributeOverride`, you must use the `value.` prefix to override properties of the embeddable object used in the map value and the `key.` prefix to override properties of the embeddable object used in the map key.

```
@Entity
public class User {
   @ElementCollection
   @AttributeOverrides({
      @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
      @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
   })
```

```
    public Map<Address,Rating> getFavHomes() { ... }
```

Note

We recommend you to migrate from `@org.hibernate.annotations.CollectionOfElements` to the new `@ElementCollection` annotation.

2.2.5.3.4. Indexed collections (List, Map)

Lists can be mapped in two different ways:

- as ordered lists, the order is not materialized in the database
- as indexed lists, the order is materialized in the database

To order lists in memory, add `@javax.persistence.OrderBy` to your property. This annotation takes into parameter a list of comma separated properties (of the target entity) and order the collection accordingly (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by the primary key of the target entity.

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
```

```
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

To store the index value in a dedicated column, use the `@javax.persistence.OrderColumn` annotation on your property. This annotations describes the column name and attributes of the column keeping the index value. This column is hosted on the table containing the association foreign key. If the column name is not specified, the default is the name of the referencing property, followed by underscore, followed by `ORDER` (in the following example, it would be `orders_ORDER`).

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderColumn(name"orders_index")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|--------------| |----------|
| Order        | | Customer |
|--------------| |----------|
| id           | | id       |
| number       | |----------|
| customer_id  |
| orders_index |
|--------------|
```

Note

We recommend you to convert `@org.hibernate.annotations.IndexColumn` usages to `@OrderColumn` unless you are making use of the base property. The `base` property lets you define the index value of the first element (aka as base index). The usual value is `0` or `1`. The default is 0 like in Java.

Likewise, maps can borrow their keys from one of the associated entity properties or have dedicated columns to store an explicit key.

To use one of the target entity property as a key of the map, use `@MapKey(name="myProperty")` (`myProperty` is a property name in the target entity). When using `@MapKey` (without property name), the target entity primary key is used. The map key uses the same column as the property pointed out: there is no additional column defined to hold the map key, and it does make sense since the map key actually represent a target property. Be aware that once loaded, the key is no longer kept in sync with the property, in other words, if you change the property value, the key will not change automatically in your Java model.

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @MapKey(name"number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> order) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

Otherwise, the map key is mapped to a dedicated column or columns. To customize things, use one of the following annotations:

- `@MapKeyColumn` if the map key is a basic type, if you don't specify the column name, the name of the property followed by underscore followed by `KEY` is used (for example `orders_KEY`).
- `@MapKeyEnumerated` / `@MapKeyTemporal` if the map key type is respectively an enum or a `Date`.
- `@MapKeyJoinColumn`/`@MapKeyJoinColumns` if the map key type is another entity.
- `@AttributeOverride`/`@AttributeOverrides` when the map key is a embeddable object. Use `key.` as a prefix for your embeddable object property names.

You can also use `@MapKeyClass` to define the type of the key if you don't use generics (at this stage, you should wonder why at this day and age you don't use generics).

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany @JoinTable(name="Cust_Order")
   @MapKeyColumn(name"orders_number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> orders) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------| |---------------|
| Order       | | Customer | | Cust_Order    |
|-------------| |----------| |---------------|
| id          | | id       | | customer_id   |
| number      | |----------| | order_id      |
| customer_id |              | orders_number |
|-------------|              |---------------|
```

Note

We recommend you to migrate from `@org.hibernate.annotations.MapKey` / `@org.hibernate.annotation.MapKeyManyToMany` to the new standard approach described above.

Let's now explore the various collection semantics based on the mapping you are choosing.

Table 2.1. Collections semantics

Semantic	java representation	annotations
Bag semantic	java.util.List, java.util.Collection	@ElementCollection or @OneToMany or @ManyToMany
Bag semantic with primary key (without the limitations of Bag semantic)	java.util.List, java.util.Collection	(@ElementCollection or @OneToMany or @ManyToMany) and @CollectionId
List semantic	java.util.List	(@ElementCollection or @OneToMany or @ManyToMany) and (@OrderColumn or @org.hibernate.annotations.IndexColumn)
Set semantic	java.util.Set	@ElementCollection or @OneToMany or @ManyToMany
Map semantic	java.util.Map	(@ElementCollection or @OneToMany or @ManyToMany) and ((nothing or @MapKeyJoinColumn/ @MapKeyColumn for true map support) OR @javax.persistence.MapKey)

Specifically, java.util.List collections without @OrderColumn or @IndexColumn are going to be considered as bags.

More support for collections are available via Hibernate specific extensions (see *Section 2.4, “Hibernate Annotation Extensions”*).

2.2.5.4. Transitive persistence with cascading

You probably have noticed the `cascade` attribute taking an array of `CascadeType` as a value. The cascade concept in JPA is very is similar to the transitive persistence and cascading of operations in Hibernate, but with slightly different semantics and cascading types:

- `CascadeType.PERSIST`: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- `CascadeType.MERGE`: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- `CascadeType.REMOVE`: cascades the remove operation to associated entities if delete() is called
- `CascadeType.REFRESH:` cascades the refresh operation to associated entities if refresh() is called
- `CascadeType.DETACH:` cascades the detach operation to associated entities if detach() is called
- `CascadeType.ALL`: all of the above

Note

CascadeType.ALL also covers Hibernate specific operations like save-update, lock etc... Check *Section 2.4.7, “Cascade”* for more information

Please refer to the chapter 6.3 of the JPA specification for more information on cascading and create/merge semantics.

You can also enable the orphan removal semantic. If an entity is removed from a `@OneToMany` collection or an associated entity is dereferenced from a `@OneToOne` association, this associated entity can be marked for deletion if `orphanRemoval` is set to true. In a way, it means that the associated entity's lifecycle is bound to the owning entity just like an embeddable object is.

```
@Entity class Customer {
   @OneToMany(orphanRemoval=true) public Set<Order> getOrders() { return orders; }
   public void setOrders(Set<Order> orders) { this.orders = orders; }
   private Set<Order> orders;

   [...]
}

@Entity class Order { ... }
```

```
Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

2.2.5.5. Association fetching

You have the ability to either eagerly or lazily fetch associated entities. The `fetch` parameter can be set to `FetchType.LAZY` or `FetchType.EAGER`. `EAGER` will try to use an outer join select to retrieve the associated object, while `LAZY` will only trigger a select when the associated object is accessed for the first time. `@OneToMany` and `@ManyToMany` associations are defaulted to `LAZY` and `@OneToOne` and `@ManyToOne` are defaulted to `EAGER`. For more information about static fetching, check *Section 2.4.5.1, “Lazy options and fetching modes”*.

The recommanded approach is to use `LAZY` on all static fetching definitions and override this choice dynamically through JP-QL. JP-QL has a `fetch` keyword that allows you to override laziness when doing a particular query. This is very useful to improve performance and is decided on a use case to use case basis.

2.2.6. Mapping composite primary keys and foreign keys to composite primary keys

Composite primary keys use a embedded class as the primary key representation, so you'd use the `@Id` and `@Embeddable` annotations. Alternatively, you can use the `@EmbeddedId` annotation. Note that the dependent class has to be serializable and implements `equals()`/`hashCode()`. You can also use `@IdClass`. These are more detailed in *Section 2.2.3, “Mapping identifier properties”*.

```
@Entity
public class RegionalArticle implements Serializable {

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

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

or alternatively

```
@Entity
public class RegionalArticle implements Serializable {

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

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

`@Embeddable` inherit the access type of its owning entity unless `@Access` is used. Composite foreign keys (if not using the default sensitive values) are defined on associations using the `@JoinColumns` element, which is basically an array of `@JoinColumn`. It is considered a good practice to express `referencedColumnNames` explicitly. Otherwise, Hibernate will suppose that you use the same order of columns as in the primary key declaration.

```
@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; //unidirectional
    ...
}
```

```
@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; //unidirectional
}
```

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

Note the explicit usage of the `referencedColumnName`.

2.2.7. Mapping secondary tables

You can map a single entity to several tables using the `@SecondaryTable` or `@SecondaryTables` class level annotations. To express that a column is in a particular table, use the `table` parameter of `@Column` or `@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;
    }
}
```

In this example, `name` will be in `MainCat`. `storyPart1` will be in `Cat1` and `storyPart2` will be in `Cat2`. `Cat1` will be joined to `MainCat` using the `cat_id` as a foreign key, and `Cat2` using `id` (ie the same column name, the `MainCat` id column has). Plus a unique constraint on `storyPart2` has been set.

2.2.8. Caching entities

Hibernate offers naturally a first level cache for entities called a persistence context via the notion of `Session`. This cache is contextual to the use case at hand. Some entities however are shared by many different use cases and are barely changed. You can cache these in what is called the second level cache.

By default, entities are not part of the second level cache. While we do not recommend that, you can override this by setting the `shared-cache-mode` element in your persistence.xml file or by using the `javax.persistence.sharedCache.mode property`. The following values are possible:

- `ENABLE_SELECTIVE` (Default and recommended value): entities are not cached unless explicitly marked as cacheable.
- `DISABLE_SELECTIVE`: entities are cached unless explicitly marked as not cacheable.
- `ALL`: all entities are always cached even if marked as non cacheable.
- `NONE`: no entity are cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The cache concurrency strategy used by default can be set with the `hibernate.cache.default_cache_concurrency_strategy` property:

- `read-only`
- `read-write`
- `nonstrict-read-write`
- `transactional`

Note

It is recommended to define the cache concurrency strategy per entity rather than using a global one. Use the `@org.hibernate.annotations.Cache` annotation for that.

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

Hibernate also let's you cache the content of a collection or the identifiers if the collection contains other entities. Use the `@Cache` annotation on the collection property.

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

`@org.hibernate.annotations.Cache` defines the caching strategy and region of a given second level cache.

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

(1) usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)

(2) region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)

(3) `include` (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

2.3. Mapping Queries

While you can write queries in your code, it is considered a good practice to externalize them:

- it make developer/DBA communications easier
- named queries are pre-compiled by Hibernate at startup time

Unfortunately, you lose the type-safety of queries written using the Criteria API.

2.3.1. Mapping JP-QL/HQL queries

You can map JP-QL/HQL queries using annotations. `@NamedQuery` and `@NamedQueries` can be defined at the class level or in a JPA XML deployment descriptor. However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

```
<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();
        ...
    }
    ...
}
```

You can also provide some hints to a query through an array of `QueryHint` through a `hints` attribute.

The available Hibernate hints are

Table 2.2. Query hints

hint	description
org.hibernate.cacheable	Whether the query should interact with the second level cache (defualt to false)
org.hibernate.cacheRegion	Cache region name (default used otherwise)
org.hibernate.timeout	Query timeout
org.hibernate.fetchSize	resultset fetch size
org.hibernate.flushMode	Flush mode used for this query
org.hibernate.cacheMode	Cache mode used for this query
org.hibernate.readOnly	Entities loaded by this query should be in read only mode or not (default to false)
org.hibernate.comment	Query comment added to the generated SQL

You can also define the lock mode by which the returned entities should be locked using the `lockMode` property. This is equivalent to the optional lock mode of the entitymanager lookup operations.

2.3.2. Mapping native queries

You can also map a native query (ie a plain SQL query). To achieve that, you need to describe the SQL resultset structure using `@SqlResultSetMapping` (or `@SqlResultSetMappings` if you plan to define several resulset mappings). Like `@NamedQuery`, a `@SqlResultSetMapping` can be defined at class level or in a JPA XML file. However its scope is global to the application.

As we will see, a `resultSetMapping` parameter is defined in `@NamedNativeQuery`, it represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property.

```
@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=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")
    })
    }
)
```

In the above example, the `night&area` named query use the `joinMapping` result set mapping. This mapping returns 2 entities, `Night` and `Area`, each property is declared and associated to a column name, actually the column name retrieved by the query. Let's now see an implicit declaration of the property / column.

```
@Entity
@SqlResultSetMapping(name="implicit",
                     entities=@EntityResult(entityClass=SpaceShip.class))
@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;
    }
}
```

In this example, we only describe the entity member of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the `model` property is bound to the `model_txt` column. If the association to a related entity involve a composite primary key, a `@FieldResult` element should be used for each foreign key column. The `@FieldResult` name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field or property of the primary key.

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=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 as surface from SpaceShip",
    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;
    }
}
```

If you retrieve a single entity and if you use the default mapping, you can use the `resultClass` attribute instead of `resultSetMapping`:

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

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the `@SqlResultsetMapping` through `@ColumnResult`. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

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

An other query hint specific to native queries has been introduced: `org.hibernate.callable` which can be true or false depending on whether the query is a stored procedure or not.

2.4. Hibernate Annotation Extensions

Hibernate 3.1 offers a variety of additional annotations that you can mix/match with your EJB 3 entities. They have been designed as a natural extension of EJB3 annotations.

To empower the EJB3 capabilities, hibernate provides specific annotations that match hibernate features. The `org.hibernate.annotations` package contains all these annotations extensions.

2.4.1. Entity

You can fine tune some of the actions done by Hibernate on entities beyond what the EJB3 spec offers.

`@org.hibernate.annotations.Entity` adds additional metadata that may be needed beyond what is defined in the standard `@Entity`

- mutable: whether this entity is mutable or not
- dynamicInsert: allow dynamic SQL for inserts
- dynamicUpdate: allow dynamic SQL for updates
- selectBeforeUpdate: Specifies that Hibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified.
- polymorphism: whether the entity polymorphism is of PolymorphismType.IMPLICIT (default) or PolymorphismType.EXPLICIT
- optimisticLock: optimistic locking strategy (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY or OptimisticLockType.ALL)

Note

@javax.persistence.Entity is still mandatory, @org.hibernate.annotations.Entity is not a replacement.

Here are some additional Hibernate annotation extensions

`@org.hibernate.annotations.BatchSize` allows you to define the batch size when fetching instances of this entity (eg. `@BatchSize(size=4)`). When loading a given entity, Hibernate will then load all the uninitialized entities of the same type in the persistence context up to the batch size.

`@org.hibernate.annotations.Proxy` defines the laziness attributes of the entity. lazy (default to true) define whether the class is lazy or not. proxyClassName is the interface used to generate the proxy (default is the class itself).

`@org.hibernate.annotations.Where` defines an optional SQL WHERE clause used when instances of this class is retrieved.

`@org.hibernate.annotations.Check` defines an optional check constraints defined in the DDL statetement.

`@OnDelete(action=OnDeleteAction.CASCADE)` on joined subclasses: use a SQL cascade delete on deletion instead of the regular Hibernate mechanism.

`@Table(appliesTo="tableName", indexes = { @Index(name="index1", columnNames={"column1", "column2"} ) } )` creates the defined indexes on the columns of table `tableName`. This can be applied on the primary table or any secondary table. The `@Tables` annotation allows your to apply indexes on different tables. This annotation is expected where `@javax.persistence.Table` or `@javax.persistence.SecondaryTable`(s) occurs.

Note

`@org.hibernate.annotations.Table` is a complement, not a replacement to `@javax.persistence.Table`. Especially, if you want to change the default name of a table, you must use `@javax.persistence.Table`, not `@org.hibernate.annotations.Table`.

`@org.hibernate.annotations.Table` can also be used to define the following elements of secondary tables:

- `fetch`: If set to JOIN, the default, Hibernate will use an inner join to retrieve a secondary table defined by a class or its superclasses and an outer join for a secondary table defined by a subclass. If set to select then Hibernate will use a sequential select for a secondary table defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a secondary defined by the class and its superclasses.

- `inverse`: If true, Hibernate will not try to insert or update the properties defined by this join. Default to false.
- `optional`: If enabled (the default), Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.
- `foreignKey`: defines the Foreign Key name of a secondary table pointing back to the primary table.

`@Immutable` marks an entity or collection as immutable. An immutable entity may not be updated by the application. This allows Hibernate to make some minor performance optimizations. Updates to an immutable entity will be ignored, but no exception is thrown. `@Immutable` must be used on root entities only. `@Immutable` placed on a collection makes the collection immutable, meaning additions and deletions to and from the collection are not allowed. A `HibernateException` is thrown in this case.

`@Persister` lets you define your own custom persistence strategy. You may, for example, specify your own subclass of `org.hibernate.persister.EntityPersister` or you might even provide a completely new implementation of the interface `org.hibernate.persister.ClassPersister` that implements persistence via, for example, stored procedure calls, serialization to flat files or LDAP.

```
@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", "length" }
@Persister(impl=MyEntityPersister.class)
public class Forest { ... }
```

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

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

2.4.2. Identifier

Hibernate Annotations goes beyond the Java Persistence specification when defining identifiers.

2.4.2.1. Generators

`@org.hibernate.annotations.GenericGenerator` and `@org.hibernate.annotations.GenericGenerators` allows you to define an Hibernate specific id generator.

```
@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` is the short name of an Hibernate3 generator strategy or the fully qualified class name of an `IdentifierGenerator` implementation. You can add some parameters through the `parameters` attribute.

Contrary to their standard counterpart, `@GenericGenerator` and `@GenericGenerators` can be used in package level annotations, making them application level generators (just like if they were in a JPA XML file).

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

2.4.2.2. @NaturalId

While not used as identifier property, some (group of) properties represent natural identifier of an entity. This is especially true when the schema uses the recommended approach of using surrogate primary key even if a natural business key exists. Hibernate allows to map such natural properties and reuse them in a `Criteria` query. The natural identifier is composed of all the properties marked `@NaturalId`.

```
@Entity
public class Citizen {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstname;
    private String lastname;

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}



//and later on query
List results = s.createCriteria( Citizen.class )
                .add( Restrictions.naturalId().set( "ssn", "1234" ).set( "state", ste ) )
                .list();
```

Note that the group of properties representing the natural identifier have to be unique (Hibernate will generate a unique constraint if the database schema is generated).

2.4.3. Property

2.4.3.1. Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

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

The SQL fragment can be as complex as you want and even include subselects.

2.4.3.2. Type

@org.hibernate.annotations.Type overrides the default hibernate type used: this is generally not necessary since the type is correctly inferred by Hibernate. Please refer to the Hibernate reference guide for more informations on the Hibernate types.

@org.hibernate.annotations.TypeDef and @org.hibernate.annotations.TypeDefs allows you to declare type definitions. These annotations can be placed at the class or package level. Note that these definitions are global for the session factory (even when defined at the class level). If the type is used on a single entity, you can place the definition on the entity itself. Otherwise, it is recommended to place the definition at the package level. In the example below, when Hibernate encounters a property of class PhoneNumer, it delegates the persistence strategy to the custom mapping type PhoneNumberType. However, properties belonging to other classes, too, can delegate their persistence strategy to PhoneNumberType, by explicitly using the @Type annotation.

Note

Package level annotations are placed in a file named package-info.java in the appropriate package. Place your annotations before the package declaration.

```
@TypeDef(
   name = "phoneNumber",
   defaultForType = PhoneNumber.class,
   typeClass = PhoneNumberType.class
)

@Entity
public class ContactDetails {
   [...]
   private PhoneNumber localPhoneNumber;
   @Type(type="phoneNumber")
   private OverseasPhoneNumber overseasPhoneNumber;
   [...]
}
```

The following example shows the usage of the parameters attribute to customize the TypeDef.

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

```
)
package org.hibernate.test.annotations.entity;

//in org/hibernate/test/annotations/entity/Forest.java
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The `@Columns` has been introduced for that purpose.

```
@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;
    ...
}
```

2.4.3.3. Index

You can define an index on a particular column using the `@Index` annotation on a one column property, the columnNames attribute will then be ignored

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

2.4.3.4. @Parent

When inside an embeddable object, you can define one of the properties as a pointer back to the owner element.

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

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


person == person.address.owner
```

2.4.3.5. Generated properties

Some properties are generated at insert or update time by your database. Hibernate can deal with such properties and triggers a subsequent select to read these properties.

```
@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;
}
```

Annotate your property as `@Generated` You have to make sure your insertability or updatability does not conflict with the generation strategy you have chosen. When `GenerationTime.INSERT` is chosen, the property must not contains insertable columns, when `GenerationTime.ALWAYS` is chosen, the property must not contains insertable nor updatable columns.

`@Version` properties cannot be `@Generated(INSERT)` by design, it has to be either `NEVER` or `ALWAYS`.

2.4.3.6. @Target

Sometimes, the type guessed by reflection is not the one you want Hibernate to use. This is especially true on components when an interface is used. You can use `@Target` to by pass the reflection guessing mechanism (very much like the `targetEntity` attribute available on associations.

```
    @Embedded
    @Target(OwnerImpl.class)
    public Owner getOwner() {
        return owner;
    }
```

2.4.3.7. Optimistic lock

It is sometimes useful to avoid increasing the version number even if a given property is dirty (particularly collections). You can do that by annotating the property (or collection) with `@OptimisticLock(excluded=true)`.

More formally, specifies that updates to this property do not require acquisition of the optimistic lock.

2.4.4. Inheritance

SINGLE_TABLE is a very powerful strategy but sometimes, and especially for legacy systems, you cannot add an additional discriminator column. For that purpose Hibernate has introduced the notion of discriminator formula: `@DiscriminatorFormula` is a replacement of `@DiscriminatorColumn` and use a SQL fragment as a formula for discriminator resolution (no need to have a dedicated column).

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

By default, when querying the top entities, Hibernate does not put a restriction clause on the discriminator column. This can be inconvenient if this column contains values not mapped in your hierarchy (through `@DiscriminatorValue`). To work around that you can use `@ForceDiscriminator` (at the class level, next to `@DiscriminatorColumn`). Hibernate will then list the available values when loading the entities.

You can define the foreign key name generated by Hibernate for subclass tables in the JOINED inheritance strategy.

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

@Entity
@ForeignKey(name = "FK_DOCU_FILE")
public class Document extends File {
```

The foreign key from the `Document` table to the `File` table will be named `FK_DOCU_FILE`.

2.4.5. Single Association related annotations

By default, when Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised by Hibernate. This might be inconvenient for legacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the `@NotFound` annotation.

This annotation can be used on a `@OneToOne` (with FK), `@ManyToOne`, `@OneToMany` or `@ManyToMany` association.

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

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted.

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

In this case Hibernate generates a cascade delete constraint at the database level.

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name by use `@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. Lazy options and fetching modes

JPA comes with the `fetch` option to define lazy loading and fetching modes, however Hibernate has a much more option set in this area. To fine tune the lazy loading and fetching strategies, some additional annotations have been introduced:

- `@LazyToOne`: defines the lazyness option on `@ManyToOne` and `@OneToOne` associations. `LazyToOneOption` can be `PROXY` (ie use a proxy based lazy loading), `NO_PROXY` (use a bytecode enhancement based lazy loading - note that build time bytecode processing is necessary) and `FALSE` (association not lazy)

- `@LazyCollection`: defines the lazyness option on `@ManyTo`Many and `@OneToMany` associations. LazyCollectionOption can be `TRUE` (the collection is lazy and will be loaded when its state is accessed), `EXTRA` (the collection is lazy and all operations will try to avoid the collection loading, this is especially useful for huge collections when loading all the elements is not necessary) and FALSE (association not lazy)
- `@Fetch`: defines the fetching strategy used to load the association. `FetchMode` can be `SELECT` (a select is triggered when the association needs to be loaded), `SUBSELECT` (only available for collections, use a subselect strategy - please refers to the Hibernate Reference Documentation for more information) or `JOIN` (use a SQL JOIN to load the association while loading the owner entity). `JOIN` overrides any lazy attribute (an association loaded through a `JOIN` strategy cannot be lazy).

The Hibernate annotations overrides the EJB3 fetching options.

Table 2.3. Lazy and fetch options equivalent

Annotations	Lazy	Fetch
@[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)

2.4.5.2. @Any

The `@Any` annotation defines a polymorphic association to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

The `@Any` annotation describes the column holding the metadata information. To link the value of the metadata information and an actual entity type, The `@AnyDef` and `@AnyDefs` annotations are used.

```
    @Any( metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @AnyMetaDef(
        idType = "integer",
        metaType = "string",
        metaValues = {
            @MetaValue( value = "S", targetEntity = StringProperty.class ),
            @MetaValue( value = "I", targetEntity = IntegerProperty.class )
        } )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

`idType` represents the target entities identifier property type and `metaType` the metadata type (usually String).

Note that `@AnyDef` can be mutualized and reused. It is recommended to place it as a package metadata in this case.

```
//on a package
@AnyMetaDef( name="property"
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
package org.hibernate.test.annotations.any;


//in a class
    @Any( metaDef="property", metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

2.4.6. Collection related annotations

2.4.6.1. Enhance collection settings

It is possible to set

- the batch size for collections using @BatchSize
- the where clause, using @Where (applied on the target entity) or @WhereJoinTable (applied on the association table)
- the check clause, using @Check
- the SQL order by clause, using @OrderBy
- the delete cascade strategy through @OnDelete(action=OnDeleteAction.CASCADE)

- the collection immutability using @Immutable: if set specifies that the elements of the collection never change (a minor performance optimization in some cases)
- a custom collection persister (ie the persistence strategy used) using `@Persister`: the class must implement `org.hibernate.persister.collectionCollectionPersister`

You can also declare a sort comparator. Use the `@Sort` annotation. Expressing the comparator type you want between unsorted, natural or custom comparator. If you want to use your own comparator implementation, you'll also have to express the implementation class using the `comparator` attribute. Note that you need to use either a `SortedSet` or a `SortedMap` interface.

```
    @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;
    }
```

Please refer to the previous descriptions of these annotations for more informations.

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name by use `@ForeignKey`. Note that this annotation has to be placed on the owning side of the relationship, `inverseName` referencing to the other side constraint.

```
@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. Extra collection types

2.4.6.2.1. Bidirectional association with indexed collections

A bidirectional association where one end is an indexed collection (ie. represented as a `@OrderColumn`, or as a Map) requires special consideration. If a property on the associated class explicitly maps the indexed value, the use of `mappedBy` is permitted:

```
@Entity
```

```
public class Parent {
    @OneToMany(mappedBy="parent")
    @OrderColumn(name="order")
    private List<Child> children;
    ...
}

@Entity
public class Child {
    ...
    //the index column is mapped as a property in the associated entity
    @Column(name="order")
    private int order;

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

But, if there is no such property on the child class, we can't think of the association as truly bidirectional (there is information available at one end of the association that is not available at the other end: the index). In this case, we can't map the collection as `mappedBy`. Instead, we could use the following mapping:

```
@Entity
public class Parent {
    @OneToMany
    @OrderColumn(name="order")
    @JoinColumn(name="parent_id", nullable=false)
    private List<Child> children;
    ...
}

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

Note that in this mapping, the collection-valued end of the association is responsible for updating the foreign key.

2.4.6.2.2. Bag with primary key

Another interesting feature is the ability to define a surrogate primary key to a bag collection. This remove pretty much all of the drawbacks of bags: update and removal are efficient, more than one `EAGER` bag per query or per entity. This primary key will be contained in a additional column of your collection table but will not be visible to the Java application. @CollectionId is used to mark a collection as id bag, it also allow to override the primary key column(s), the primary key type and the generator strategy. The strategy can be `identity`, or any defined generator name of your 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();
    ...
}
```

2.4.6.2.3. @ManyToAny

`@ManyToAny` allows polymorphic associations to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

```
    @ManyToAny(
            metaColumn = @Column( name = "property_type" ) )
    @AnyMetaDef(
        idType = "integer",
        metaType = "string",
        metaValues = {
            @MetaValue( value = "S", targetEntity = StringProperty.class ),
            @MetaValue( value = "I", targetEntity = IntegerProperty.class ) } )
    @Cascade( { org.hibernate.annotations.CascadeType.ALL } )
    @JoinTable( name = "obj_properties", joinColumns = @JoinColumn( name = "obj_id" ),
            inverseJoinColumns = @JoinColumn( name = "property_id" ) )
    public List<Property> getGeneralProperties() {
```

Like `@Any`, `@ManyToAny` can use named `@AnyDefs`, see *Section 2.4.5.2, “@Any”* for more info.

2.4.7. Cascade

Hibernate offers more operations than the Java Persistence specification. You can use the `@Cascade` annotation to cascade the following operations:

- PERSIST

- MERGE
- REMOVE
- REFRESH
- DELETE
- SAVE_UPDATE
- REPLICATE
- DELETE_ORPHAN (alternatively, use the `@OneToOne.orphanRemoval` or `@OneToMany.orphanRemoval` flag)
- LOCK
- EVICT (alternatively, use the standard DETACH flag).

This is especially useful for `SAVE_UPDATE` (which is the operation cascaded at flush time if you use plain Hibernate Annotations - Hibernate EntityManager cascade `PERSIST` at flush time as per the specification).

```
@OneToMany( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
@Cascade(org.hibernate.annotations.CascadeType.REPLICATE)
public Collection<Employer> getEmployers()
```

It is recommended to use `@Cascade` to compliment `@*To*(cascade=...)` as shown in the previous example.

2.4.8. Filters

Hibernate has the ability to apply arbitrary filters on top of your data. Those filters are applied at runtime on a given session. First, you need to define them.

`@org.hibernate.annotations.FilterDef` or `@FilterDefs` define filter definition(s) used by filter(s) using the same name. A filter definition has a name() and an array of parameters(). A parameter will allow you to adjust the behavior of the filter at runtime. Each parameter is defined by a `@ParamDef` which has a name and a type. You can also define a defaultCondition() parameter for a given `@FilterDef` to set the default condition to use when none are defined in each individual `@Filter`. A `@FilterDef`(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. `@Filter` is used and placed either on the entity or the collection element

```
@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 { ... }
```

When the collection use an association table as a relational representation, you might want to apply the filter condition to the association table itself or to the target entity table. To apply the constraint on the target entity, use the regular `@Filter` annotation. However, if you wan to target the association table, use the `@FilterJoinTable` annotation.

```
    @OneToMany
    @JoinTable
    //filter on the target entity table
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    //filter on the association table
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

2.4.9. Queries

Since Hibernate has more features on named queries than the one defined in the JPA specification, `@org.hibernate.annotations.NamedQuery`, `@org.hibernate.annotations.NamedQueries`, `@org.hibernate.annotations.NamedNativeQuery` and `@org.hibernate.annotations.NamedNativeQueries` have been introduced. They add some attributes to the standard version and can be used as a replacement:

- flushMode: define the query flush mode (Always, Auto, Commit or Manual)
- cacheable: whether the query should be cached or not
- cacheRegion: cache region used if the query is cached
- fetchSize: JDBC statement fetch size for this query
- timeout: query time out
- callable: for native queries only, to be set to true for stored procedures
- comment: if comments are activated, the comment seen when the query is sent to the database.
- cacheMode: Cache interaction mode (get, ignore, normal, put or refresh)
- readOnly: whether or not the elements retrievent from the query are in read only mode.

Those hints can be set in a standard `@javax.persistence.NamedQuery` annotations through the detyped `@QueryHint`. Another key advantage is the ability to set those annotations at a package level.

2.4.10. Custom SQL for CRUD operations

Hibernate gives you the ability to override every single SQL statement generated. We have seen native SQL query usage already, but you can also override the SQL statement used to load or change the state of entities.

```
@Entity
@Table(name="CHAOS")
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")
@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")
@SQLDelete( sql="DELETE CHAOS WHERE id = ?")
@SQLDeleteAll( sql="DELETE CHAOS")
@Loader(namedQuery = "chaos")
@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from
 CHAOS where id= ?", resultClass = Chaos.class)
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`, `@SQLDeleteAll` respectively override the INSERT statement, UPDATE statement, DELETE statement, DELETE statement to remove all entities.

If you expect to call a store procedure, be sure to set the `callable` attribute to true (`@SQLInsert(callable=true, ...)`).

To check that the execution happens correctly, Hibernate allows you to define one of those three strategies:

- NONE: no check is performed: the store procedure is expected to fail upon issues
- COUNT: use of rowcount to check that the update is successful
- PARAM: like COUNT but using an output parameter rather that the standard mechanism

To define the result check style, use the `check` parameter (`@SQLUpdate(check=ResultCheckStyle.COUNT, ...)`).

You can also override the SQL load statement by a native SQL query or a HQL query. You just have to refer to a named query with the `@Loader` annotation.

You can use the exact same set of annotations to override the collection related statements.

```
@OneToMany
@JoinColumn(name="chaos_fk")
@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")
@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")
```

```
private Set<CasimirParticle> particles = new HashSet<CasimirParticle>();
```

The parameters order is important and is defined by the order Hibernate handle properties. You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations as that will override the Hibernate generated static sql.)

Overriding SQL statements for secondary tables is also possible using `@org.hibernate.annotations.Table` and either (or all) attributes `sqlInsert`, `sqlUpdate`, `sqlDelete`:

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "`Cat nbr1`"),
    @SecondaryTable(name = "Cat2"})
@org.hibernate.annotations.Tables( {
    @Table(appliesTo = "Cat", comment = "My cat table" ),
    @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
public class Cat implements Serializable {
```

The previous example also show that you can give a comment to a given table (promary or secondary): This comment will be used for DDL generation.

2.4.11. Tuplizer

`org.hibernate.tuple.Tuplizer`, and its sub-interfaces, are responsible for managing a particular representation of a piece of data, given that representation's `org.hibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing which knows how to create such a data structure and how to extract values from and inject values into such a data structure. For example, for the POJO entity mode, the correpsonding tuplizer knows how create the POJO through its constructor and how to access the POJO properties using the defined property accessors. There are two high-level types of Tuplizers, represented by the `org.hibernate.tuple.EntityTuplizer` and `org.hibernate.tuple.ComponentTuplizer` interfaces. EntityTuplizers are responsible for managing the above mentioned contracts in regards to entities, while `ComponentTuplizers` do the same for components. Check the Hibernate reference documentation for more information.

To define tuplixer in annotations, simply use the `@Tuplizer` annotation on the according element

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);

    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

2.4.12. Fetch profiles

In *Section 2.4.5.1, “Lazy options and fetching modes”* we have seen how to affect the fetching strategy for associated objects using the `@Fetch` annotation. An alternative approach is a so called fetch profile. A fetch profile is a named configuration associated with the `org.hibernate.SessionFactory` which gets enabled on the `org.hibernate.Session`. Once enabled on a `org.hibernate.Session`, the fetch profile will be in affect for that session until it is explicitly disabled. Lets look at an example:

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {
   @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   private long customerNumber;

   @OneToMany
   private Set<Order> orders;

   // standard getter/setter
   ...
}
```

In the normal case the orders association would be lazy loaded by Hibernate, but in a usecase where it is more efficient to load the customer and their orders together you could do something like this:

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" );  // name matches @FetchProfile name
Customer customer = (Customer) session.get( Customer.class, customerId );
session.disableFetchProfile( "customer-with-orders" ); // or just close the session
...
```

Note

Fetch profile definitions are global and it does not matter on which class you place them. You can place the `@FetchProfile` annotation either onto a class or package (package-info.java). In order to define multiple fetch profiles for the same class or package `@FetchProfiles` can be used.

Currently only join style fetch profiles are supported, but they plan is to support additional styles. See *HHH-3414* [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414] for details. Refer also to the discussion about fetch profiles in the Hibernate Core documentation.

Overriding metadata through XML

The primary target for metadata in EJB3 is annotations, but the EJB3 specification provides a way to override or replace the annotation defined metadata through an XML deployment descriptor. In the current release only pure EJB3 annotations overriding are supported. If you wish to use Hibernate specific features in some entities, you'll have to either use annotations or fallback to hbm files. You can of course mix and match annotated entities and entities describes in hbm files.

The unit test suite shows some additional XML file samples.

3.1. Principles

The XML deployment descriptor structure has been designed to reflect the annotations one. So if you know the annotations structure, using the XML schema will be straightforward for you.

You can define one or more XML files describing your metadata, these files will be merged by the overriding engine.

3.1.1. Global level metadata

You can define global level metadata available for all XML files. You must not define these metadata more than once per deployment.

```
<?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_2_0.xsd"
  version="2.0">

    <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` means that all entity, mapped-superclasses and embeddable metadata should be picked up from XML (ie ignore annotations).

`schema / catalog` will override all default definitions of schema and catalog in the metadata (both XML and annotations).

`cascade-persist` means that all associations have PERSIST as a cascade type. We recommend you to not use this feature.

3.1.2. Entity level metadata

You can either define or override metadata informations on a given entity.

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

<entity-mappings                                                                   ❶
  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_2_0.xsd"
  version="2.0">

    <package>org.hibernate.test.annotations.reflection</package>                   ❷
    <entity class="Administration" access="PROPERTY" metadata-complete="true">     ❸
        <table name="tbl_admin">                                                   ❹
            <unique-constraint>
                <column-name>firstname</column-name>
                <column-name>lastname</column-name>
            </unique-constraint>
        </table>
        <secondary-table name="admin2">                                            ❺
            <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"/>                                   ❻
        <inheritance strategy="JOINED"/>                                           ❼
        <sequence-generator name="seqhilo" sequence-name="seqhilo"/>               ❽
        <table-generator name="table" table="tablehilo"/>                          ❾
        ...
    </entity>

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

❶ `entity-mappings`: entity-mappings is the root element for all XML files. You must declare the xml schema, the schema file is included in the hibernate-annotations.jar file, no internet access will be processed by Hibernate Annotations.

❷ `package` (optional): default package used for all non qualified class names in the given deployment descriptor file.

❸ `entity`: desribes an entity.

   `metadata-complete` defines whether the metadata description for this element is complete or not (in other words, if annotations present at the class level should be considered or not).

An entity has to have a `class` attribute refering the java class the metadata applies on.

You can overrides entity name through the `name` attribute, if none is defined and if an `@Entity.name` is present, then it is used (provided that metadata complete is not set).

For metadata complete (see below) element, you can define an `access` (either `FIELD` or `PROPERTY` (default)). For non medatada complete element, if `access` is not defined, the @Id position will lead position, if `access` is defined, the value is used.

4 `table`: you can declare table properties (name, schema, catalog), if none is defined, the java annotation is used.

You can define one or several unique constraints as seen in the example

5 `secondary-table`: defines a secondary table very much like a regular table except that you can define the primary key / foreign key column(s) through the `primary-key-join-column` element. On non metadata complete, annotation secondary tables are used only if there is no `secondary-table` definition, annotations are ignored otherwise.

6 `id-class`: defines the id class in a similar way `@IdClass` does

7 `inheritance`: defines the inheritance strategy (`JOINED`, `TABLE_PER_CLASS`, `SINGLE_TABLE`), Available only at the root entity level

8 `sequence-generator`: defines a sequence generator

9 `table-generator`: defines a table generator

10 `primary-key-join-column`: defines the primary key join column for sub entities when JOINED inheritance strategy is used

```
<?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_2_0.xsd"
  version="2.0">

    <package>org.hibernate.test.annotations.reflection</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`: defines the discriminator value and the column holding it when the SINGLE_TABLE inheritance strategy is chosen
2. `named-query`: defines named queries and possibly the hints associated to them. Those definitions are additive to the one defined in annotations, if two definitions have the same name, the XML one has priority.
3. `named-native-query`: defines an named native query and its sql result set mapping. Alternatively, you can define the `result-class`. Those definitions are additive to the one defined in annotations, if two definitions have the same name, the XML one has priority.
4. `sql-result-set-mapping`: describes the result set mapping structure. You can define both entity and column mappings. Those definitions are additive to the one defined in annotations, if two definitions have the same name, the XML one has priority
5. `attribute-override / association-override`: defines a column or join column overriding. This overriding is additive to the one defined in annotations

Same applies for `<embeddable>` and `<mapped-superclass>`.

3.1.3. Property level metadata

You can of course defines XML overriding for properties. If metadata complete is defined, then additional properties (ie at the Java level) will be ignored. Otherwise, once you start overriding a property, all annotations on the given property are ignored. All property level metadata behave in `entity/attributes`, `mapped-superclass/attributes` or `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>
```

You can override a property through `id`, `embedded-id`, `version`, `embedded` and `basic`. Each of these elements can have subelements accordingly: `lob`, `temporal`, `enumerated`, `column`.

3.1.4. Association level metadata

You can define XML overriding for associations. All association level metadata behave in `entity/attributes`, `mapped-superclass/attributes` or `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>
```

You can override an association through `one-to-many`, `one-to-one`, `many-to-one`, and `many-to-many`. Each of these elements can have subelements accordingly: `join-table` (which can have `join-columns` and `inverse-join-columns`), `join-columns`, `map-key`, and `order-by`. `mapped-by` and `target-entity` can be defined as attributes when it makes sense. Once again the structure is reflects the annotations structure. You can find all semantic informations in the chapter describing annotations.

Additional modules

Hibernate Annotations mainly focuses on persistence metadata. The project also have a nice integration with some external modules.

4.1. Bean Validation

Bean Validation standardizes how to define and declare domain model level constraints. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. Bean Validation can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Bean Validation and its reference implementation Hibernate Validator has been designed for that purpose.

The integration between Hibernate and Bean Validation works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts, updates and deletes done by Hibernate.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in a set of `ConstraintViolations`. Among other information, the `ConstraintViolation` contains an error description message that can embed the parameter values bundle with the annotation (eg. size limit), and message strings that may be externalized to a `ResourceBundle`.

4.1.1. Adding Bean Validation

To enable the Hibernate - Bean Validation integration, simply add a Bean Validation provider (preferably Hibernate Validation 4) in your classpath.

4.1.2. Configuration

By default, no configuration is necessary.

The `Default` group is validated on entity insert and update and the database model is updated accordingly based on the `Default` group as well.

You can customize the Bean Validation integration by setting the validation mode. Use the `javax.persistence.validation.mode` property and set it up for example in your `persistence.xml` file or your `hibernate.cfg.xml` file. Several options are possible:

- `auto` (default): enable integration between Bean Validation and Hibernate (callback and ddl generation) only if Bean Validation is present in the classpath.
- `none`: disable all integration between Bean Validation and Hibernate
- `callback`: only validate entities when they are either inserted, updated or deleted. An exception is raised if no Bean Validation provider is present in the classpath.
- `ddl`: only apply constraints to the database schema when generated by Hibernate. An exception is raised if no Bean Validation provider is present in the classpath. This value is not defined by the Java Persistence spec and is specific to Hibernate.

Note

You can use both `callback` and `ddl` together by setting the property to `callback, dll`

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.mode"
                value="callback, ddl"/>
    </properties>
  </persistence-unit>
</persistence>
```

This is equivalent to `auto` except that if no Bean Validation provider is present, an exception is raised.

If you want to validate different groups during insertion, update and deletion, use:

- `javax.persistence.validation.group.pre-persist`: groups validated when an entity is about to be persisted (default to `Default`)
- `javax.persistence.validation.group.pre-update`: groups validated when an entity is about to be updated (default to `Default`)
- `javax.persistence.validation.group.pre-remove`: groups validated when an entity is about to be deleted (default to no group)
- `org.hibernate.validator.group.ddl`: groups considered when applying constraints on the database schema (default to `Default`)

Each property accepts the fully qualified class names of the groups validated separated by a comma (,)

Example 4.1. Using custom groups for validation

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.group.Default, com.acme.group.Strict"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value="com.acme.group.OnDelete"/>
      <property name="org.hibernate.validator.group.ddl"
                value="com.acme.group.DDL"/>
    </properties>
  </persistence-unit>
</persistence>
```

Note

You can set these properties in `hibernate.cfg.xml`, `hibernate.properties` or programmatically.

4.1.3. Catching violations

If an entity is found to be invalid, the list of constraint violations is propagated by the `ConstraintViolationException` which exposes the set of `ConstraintViolation`s.

This exception is wrapped in a `RollbackException` when the violation happens at commit time. Otherwise the `ConstraintViolationException` is returned (for example when calling `flush()`. Note that generally, catchable violations are validated at a higher level (for example in Seam / JSF 2 via the JSF - Bean Validation integration or in your business layer by explicitly calling Bean Validation).

An application code will rarely be looking for a `ConstraintViolationException` raised by Hibernate. This exception should be treated as fatal and the persistence context should be discarded (`EntityManager` or `Session`).

4.1.4. Database schema

Hibernate uses Bean Validation constraints to generate an accurate database schema:

- `@NotNull` leads to a not null column (unless it conflicts with components or table inheritance)
- `@Size.max` leads to a `varchar(max)` definition for Strings
- `@Min`, `@Max` lead to column checks (like `value <= max`)
- `@Digits` leads to the definition of precision and scale (ever wondered which is which? It's easy now with `@Digits` :))

These constraints can be declared directly on the entity properties or indirectly by using constraint composition.

4.2. Hibernate Validator 3

Warning

We strongly encourage you to use Hibernate Validator 4 and the Bean Validation integration. Consider Hibernate Validator 3 as legacy.

4.2.1. Description

Annotations are a very convenient and elegant way to specify invariant constraints for a domain model. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. A validator can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Hibernate Validator has been designed for that purpose.

Hibernate Validator works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts and updates done by Hibernate. Hibernate Validator is not limited to use with Hibernate. You can easily use it anywhere in your application.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in an array of `InvalidValue` s. Among other information, the `InvalidValue` contains an error description message that can embed the parameter values bundle with the annotation (eg. length limit), and message strings that may be externalized to a `ResourceBundle` .

4.2.2. Integration with Hibernate Annotations

If Hibernate Validator (`hibernate-validator.jar`) is available in the classpath, Hibernate Annotations will integrate in two ways:

- Constraints will be applied to the Data Definition Language. In other words, the database schema will reflect the constraints (provided that you use the hbm2ddl tool).
- Before an entity change is applied to the database (insert or update), the entity is validated. Validation errors, if any, will be carried over through an `InvalidStateException`.

For entities free of validation rules, the runtime performance cost is null.

To disable constraint propagation to DDL, set up `hibernate.validator.apply_to_ddl` to false in the configuration file. Such a need is very uncommon and not recommended.

To disable pre-entity change validation, set up `hibernate.validator.autoregister_listeners` to false in the configuration file. Such a need is very uncommon and not recommended.

Check the Hibernate Validator reference documentation for more information.

4.3. Hibernate Search

4.3.1. Description

Full text search engines like Apache Lucene™ are a very powerful technology to bring free text/ efficient queries to applications. If suffers several mismatches when dealing with a object domain model (keeping the index up to date, mismatch between the index structure and the domain model, querying mismatch...) Hibernate Search indexes your domain model thanks to a few annotations, takes care of the database / index synchronization and brings you back regular managed objects from free text queries. Hibernate Search is using *Apache Lucene* [http://lucene.apache.org] under the cover.

4.3.2. Integration with Hibernate Annotations

Hibernate Search integrates with Hibernate Annotations transparently provided that hibernate-search.jar is present in the classpath. If you do not wish to automatically register Hibernate Search event listeners, you can set `hibernate.search.autoregister_listeners` to false. Such a need is very uncommon and not recommended.

Check the Hibernate Search reference documentation for more information.