Hibernate ORM 5.2.8.Final User Guide

Vlad Mihalcea, Steve Ebersole, Andrea Boriero, Gunnar Morling, Gail Badner, Chris Cranford, Emmanuel Bernard, Sanne Grinovero, Brett Meyer, Hardy Ferentschik, Gavin King, Christian Bauer, Max Rydahl Andersen, Karel Maesen, Radim Vansa, Louis Jacomet

Table of Contents

Preface

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping (http://en.wikipedia.org/wiki/Object-relational_mapping) refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa).

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, Hibernate does not hide the power of SQL from you and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

Get Involved

- Use Hibernate and report any bugs or issues you find. See Issue Tracker (http://hibernate.org/issuetracker) for details.
- Try your hand at fixing some bugs or implementing enhancements. Again, see Issue Tracker (http://hibernate.org/issuetracker).
- Engage with the community using mailing lists, forums, IRC, or other ways listed in the Community section (http://hibernate.org/community).
- Help improve or translate this documentation. Contact us on the developer mailing list if you have interest.
- Spread the word. Let the rest of your organization know about the benefits of Hibernate.

System Requirements

Hibernate 5.2 and later versions require at least Java 1.8 and JDBC 4.2.

Hibernate 5.1 and older versions require at least Java 1.6 and JDBC 4.0.

> When building Hibernate 5.1 or older from sources, you need Java 1.7 due to a bug in the JDK 1.6 compiler.

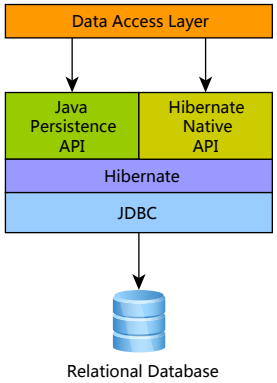
Getting Started Guide

New users may want to first look through the Hibernate Getting Started Guide (https://docs.jboss.org/hibernate/orm/5.2/quickstart/html_single/) for basic information as well as tutorials. There is also a series of topical guides (http://docs.jboss.org/hibernate/orm/5.2/topical/html_single/) providing deep dives into various topics.

> While having a strong background in SQL is not required to use Hibernate, it certainly helps a lot because it all boils down to SQL statements. Probably even more important is an understanding of data modeling principles. You might want to consider these resources as a good starting point:
>
> - Data Modeling Wikipedia definition (http://en.wikipedia.org/wiki/Data_modeling)
> - Data Modeling 101 (http://www.agiledata.org/essays/dataModeling101.html)
>
> Understanding the basics of transactions and design patterns such as *Unit of Work* PoEAA or *Application Transaction* are important as well. These topics will be discussed in the documentation, but a prior understanding will certainly help.

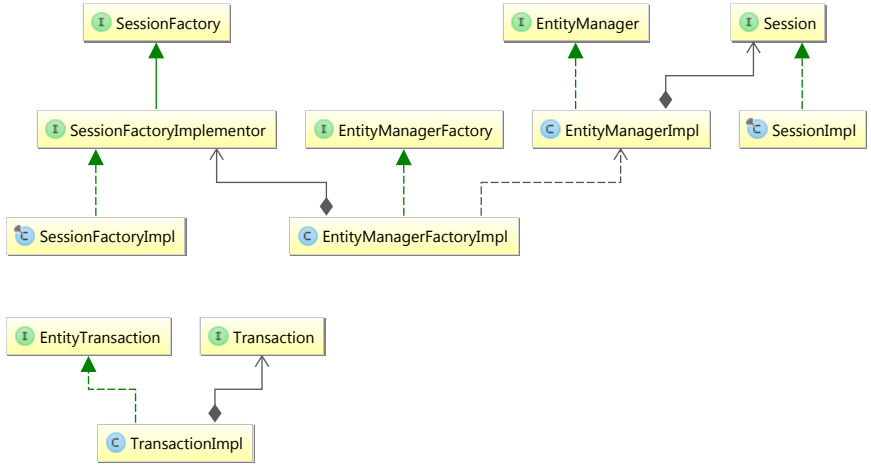
1. Architecture

1.1. Overview



Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database, as can be seen in the diagram above. The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data. Here we will introduce the essential Hibernate APIs. This will be a brief introduction; we will discuss these contracts in detail later.

As a JPA provider, Hibernate implements the Java Persistence API specifications and the association between JPA interfaces and Hibernate specific implementations can be visualized in the following diagram:



SessionFactory (`org.hibernate.SessionFactory`)

A thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for `org.hibernate.Session` instances. The `EntityManagerFactory` is the JPA equivalent of a `SessionFactory` and basically those two converge into the same `SessionFactory` implementation.

A `SessionFactory` is very expensive to create, so, for any given database, the application should have only one associated `SessionFactory`. The `SessionFactory` maintains services that Hibernate uses across all `Session(s)` such as second level caches, connection pools, transaction system integrations, etc.

Session (`org.hibernate.Session`)

A single-threaded, short-lived object conceptually modeling a "Unit of Work" PoEAA. In JPA nomenclature, the `Session` is represented by an `EntityManager`.

Behind the scenes, the Hibernate `Session` wraps a JDBC `java.sql.Connection` and acts as a factory for `org.hibernate.Transaction` instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (`org.hibernate.Transaction`)

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. `EntityTransaction` is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

2. Domain Model

The term domain model (https://en.wikipedia.org/wiki/Domain_model) comes from the realm of data modeling. It is the model that ultimately describes the problem domain (https://en.wikipedia.org/wiki/Problem_domain) you are working in. Sometimes you will also hear the term *persistent classes*.

Ultimately the application domain model is the central character in an ORM. They make up the classes you wish to map. Hibernate works best if these classes follow the Plain Old Java Object (POJO) / JavaBean programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of `java.util.Map` instances, for example).

Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose. With the coming of JPA, most of this information is now defined in a way that is portable across ORM/JPA providers using annotations (and/or standardized XML format). This chapter will focus on JPA mapping where possible. For Hibernate mapping features not supported by JPA we will prefer Hibernate extension annotations.

2.1. Mapping types

Hibernate understands both the Java and JDBC representations of application data. The ability to read/write this data from/to the database is the function of a Hibernate *type*. A type, in this usage, is an implementation of the `org.hibernate.type.Type` interface. This Hibernate type also describes various aspects of behavior of the Java type such as how to check for equality, how to

clone values, etc.

> *Usage of the word type*
>
> The Hibernate type is neither a Java type nor a SQL data type. It provides information about both of these as well as understanding marshalling between.
>
> When you encounter the term type in discussions of Hibernate, it may refer to the Java type, the JDBC type, or the Hibernate type, depending on context.

To help understand the type categorizations, let's look at a simple table and domain model that we wish to map.

Example 1. Simple table and domain model

```sql
create table Contact (
    id integer not null,
    first varchar(255),
    last varchar(255),
    middle varchar(255),
    notes varchar(255),
    starred boolean not null,
    website varchar(255),
    primary key (id)
)
```

JAVA

```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

    private String first;

    private String middle;

    private String last;

    // getters and setters omitted
}
```

In the broadest sense, Hibernate categorizes types into two groups:

- Value types
- Entity types

2.1.1. Value types

A value type is a piece of data that does not define its own lifecycle. It is, in effect, owned by an entity, which defines its lifecycle.

Looked at another way, all the state of an entity is made up entirely of value types. These state fields or JavaBean properties are termed *persistent attributes*. The persistent attributes of the `Contact` class are value types.

Value types are further classified into three sub-categories:

Basic types

in mapping the `Contact` table, all attributes except for name would be basic types. Basic types are discussed in detail in *Basic Types*

Embeddable types

the name attribute is an example of an embeddable type, which is discussed in details in *Embeddable Types*

Collection types

although not featured in the aforementioned example, collection types are also a distinct category among value types. Collection types are further discussed in *Collections*

2.1.2. Entity types

Entities, by nature of their unique identifier, exist independently of other objects whereas values do not. Entities are domain model classes which correlate to rows in a database table, using a unique identifier. Because of the requirement for a unique identifier, entities exist independently and define their own lifecycle. The `Contact` class itself would be an example of an entity.

Mapping entities is discussed in detail in *Entity*.

2.2. Naming strategies

Part of the mapping of an object model to the relational database is mapping names from the object model to the corresponding database names. Hibernate looks at this as 2 stage process:

- The first stage is determining a proper logical name from the domain model mapping. A logical name can be either explicitly specified by the user (using `@Column` or `@Table` e.g.) or it can be implicitly determined by Hibernate through an ImplicitNamingStrategy contract.
- Second is the resolving of this logical name to a physical name which is defined by the PhysicalNamingStrategy contract.

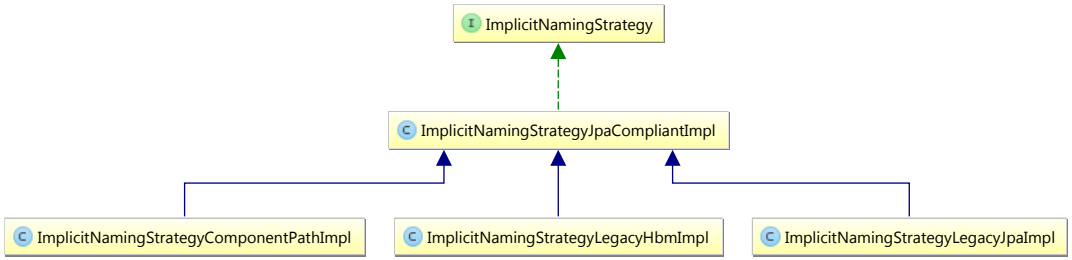
> *Historical NamingStrategy contract*
>
> Historically Hibernate defined just a single `org.hibernate.cfg.NamingStrategy`. That singular NamingStrategy contract actually combined the separate concerns that are now modeled individually as ImplicitNamingStrategy and PhysicalNamingStrategy.
>
> Also, the NamingStrategy contract was often not flexible enough to properly apply a given naming "rule", either because the API lacked the information to decide or because the API was honestly not well defined as it grew.
>
> Due to these limitation, `org.hibernate.cfg.NamingStrategy` has been deprecated and then removed in favor of ImplicitNamingStrategy and PhysicalNamingStrategy.

At the core, the idea behind each naming strategy is to minimize the amount of repetitive information a developer must provide for mapping a domain model.

> *JPA Compatibility*
>
> JPA defines inherent rules about implicit logical name determination. If JPA provider portability is a major concern, or if you really just like the JPA-defined implicit naming rules, be sure to stick with ImplicitNamingStrategyJpaCompliantImpl (the default)
>
> Also, JPA defines no separation between logical and physical name. Following the JPA specification, the logical name **is** the physical name. If JPA provider portability is important, applications should prefer not to specify a PhysicalNamingStrategy.

2.2.1. ImplicitNamingStrategy

When an entity does not explicitly name the database table that it maps to, we need to implicitly determine that table name. Or when a particular attribute does not explicitly name the database column that it maps to, we need to implicitly determine that column name. There are examples of the role of the `org.hibernate.boot.model.naming.ImplicitNamingStrategy` contract to determine a logical name when the mapping did not provide an explicit name.



Hibernate defines multiple ImplicitNamingStrategy implementations out-of-the-box. Applications are also free to plug-in custom implementations.

There are multiple ways to specify the ImplicitNamingStrategy to use. First, applications can specify the implementation using the `hibernate.implicit_naming_strategy` configuration setting which accepts:

- pre-defined "short names" for the out-of-the-box implementations

  `default`
  for `org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl` - an alias for `jpa`

  `jpa`

for `org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl` - the JPA 2.0 compliant naming strategy

`legacy-hbm`

for `org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyHbmImpl` - compliant with the original Hibernate NamingStrategy

`legacy-jpa`

for `org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl` - compliant with the legacy NamingStrategy developed for JPA 1.0, which was unfortunately unclear in many respects regarding implicit naming rules

`component-path`

for `org.hibernate.boot.model.naming.ImplicitNamingStrategyComponentPathImpl` - mostly follows `ImplicitNamingStrategyJpaCompliantImpl` rules, except that it uses the full composite paths, as opposed to just the ending property part

- reference to a Class that implements the `org.hibernate.boot.model.naming.ImplicitNamingStrategy` contract
- FQN of a class that implements the `org.hibernate.boot.model.naming.ImplicitNamingStrategy` contract

Secondly, applications and integrations can leverage `org.hibernate.boot.MetadataBuilder#applyImplicitNamingStrategy` to specify the ImplicitNamingStrategy to use. See Bootstrap for additional details on bootstrapping.

2.2.2. PhysicalNamingStrategy

Many organizations define rules around the naming of database objects (tables, columns, foreign-keys, etc). The idea of a PhysicalNamingStrategy is to help implement such naming rules without having to hard-code them into the mapping via explicit names.

While the purpose of an ImplicitNamingStrategy is to determine that an attribute named `accountNumber` maps to a logical column name of `accountNumber` when not explicitly specified, the purpose of a PhysicalNamingStrategy would be, for example, to say that the physical column name should instead be abbreviated `acct_num`.

> It is true that the resolution to `acct_num` could have been handled in an ImplicitNamingStrategy in this case. But the point is separation of concerns. The PhysicalNamingStrategy will be applied regardless of whether the attribute explicitly specified the column name or whether we determined that implicitly. The ImplicitNamingStrategy would only be applied if an explicit name was not given. So it depends on needs and intent.

The default implementation is to simply use the logical name as the physical name. However applications and integrations can define custom implementations of this PhysicalNamingStrategy contract. Here is an example PhysicalNamingStrategy for a fictitious company named Acme Corp whose naming standards are to:

- prefer underscore-delimited words rather than camel-casing
- replace certain words with standard abbreviations

Example 2. Example PhysicalNamingStrategy implementation

JAVA

```java
/*
 * Hibernate, Relational Persistence for Idiomatic Java
 *
 * License: GNU Lesser General Public License (LGPL), version 2.1 or later.
 * See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
 */
package org.hibernate.userguide.naming;

import java.util.LinkedList;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.TreeMap;

import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategy;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

import org.apache.commons.lang3.StringUtils;

/**
 * An example PhysicalNamingStrategy that implements database object naming standards
 * for our fictitious company Acme Corp.
 * <p/>
 * In general Acme Corp prefers underscore-delimited words rather than camel casing.
 * <p/>
 * Additionally standards call for the replacement of certain words with abbreviations.
 *
 * @author Steve Ebersole
 */
public class AcmeCorpPhysicalNamingStrategy implements PhysicalNamingStrategy {
	private static final Map<String,String> ABBREVIATIONS = buildAbbreviationMap();

	@Override
	public Identifier toPhysicalCatalogName(Identifier name, JdbcEnvironment jdbcEnvironment) {
		// Acme naming standards do not apply to catalog names
		return name;
	}

	@Override
	public Identifier toPhysicalSchemaName(Identifier name, JdbcEnvironment jdbcEnvironment) {
		// Acme naming standards do not apply to schema names
		return null;
	}

	@Override
	public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment jdbcEnvironment) {
		final List<String> parts = splitAndReplace( name.getText() );
		return jdbcEnvironment.getIdentifierHelper().toIdentifier(
				join( parts ),
				name.isQuoted()
		);
	}

	@Override
	public Identifier toPhysicalSequenceName(Identifier name, JdbcEnvironment jdbcEnvironment) {
```

```
        final LinkedList<String> parts = splitAndReplace( name.getText() );
        // Acme Corp says all sequences should end with _seq
        if ( !"seq".equalsIgnoreCase( parts.getLast() ) ) {
                parts.add( "seq" );
        }
        return jdbcEnvironment.getIdentifierHelper().toIdentifier(
                        join( parts ),
                        name.isQuoted()
        );
}

@Override
public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment jdbcEnvironment) {
        final List<String> parts = splitAndReplace( name.getText() );
        return jdbcEnvironment.getIdentifierHelper().toIdentifier(
                        join( parts ),
                        name.isQuoted()
        );
}

private static Map<String, String> buildAbbreviationMap() {
        TreeMap<String,String> abbreviationMap = new TreeMap<> ( String.CASE_INSENSITIVE_ORDER );
        abbreviationMap.put( "account", "acct" );
        abbreviationMap.put( "number", "num" );
        return abbreviationMap;
}

private LinkedList<String> splitAndReplace(String name) {
        LinkedList<String> result = new LinkedList<>();
        for ( String part : StringUtils.splitByCharacterTypeCamelCase( name ) ) {
                if ( part == null || part.trim().isEmpty() ) {
                        // skip null and space
                        continue;
                }
                part = applyAbbreviationReplacement( part );
                result.add( part.toLowerCase( Locale.ROOT ) );
        }
        return result;
}

private String applyAbbreviationReplacement(String word) {
        if ( ABBREVIATIONS.containsKey( word ) ) {
                return ABBREVIATIONS.get( word );
        }

        return word;
}

private String join(List<String> parts) {
        boolean firstPass = true;
        String separator = "";
        StringBuilder joined = new StringBuilder();
        for ( String part : parts ) {
                joined.append( separator ).append( part );
                if ( firstPass ) {
                        firstPass = false;
                        separator = "_";
                }
```

```
                }
                return joined.toString();
            }
        }
```

There are multiple ways to specify the PhysicalNamingStrategy to use. First, applications can specify the implementation using the `hibernate.physical_naming_strategy` configuration setting which accepts:

- reference to a Class that implements the `org.hibernate.boot.model.naming.PhysicalNamingStrategy` contract
- FQN of a class that implements the `org.hibernate.boot.model.naming.PhysicalNamingStrategy` contract

Secondly, applications and integrations can leverage `org.hibernate.boot.MetadataBuilder#applyPhysicalNamingStrategy`. See Bootstrap for additional details on bootstrapping.

2.3. Basic Types

Basic value types usually map a single database column, to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which follow the natural mappings recommended by the JDBC specifications.

Internally Hibernate uses a registry of basic types when it needs to resolve a specific `org.hibernate.type.Type`.

2.3.1. Hibernate-provided BasicTypes

Table 1. Standard BasicTypes

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
StringType	VARCHAR	java.lang.String	string, java.lang.String
MaterializedClob	CLOB	java.lang.String	materialized_clob
TextType	LONGVARCHAR	java.lang.String	text
CharacterType	CHAR	char, java.lang.Character	char, java.lang.Character
BooleanType	BIT	boolean, java.lang.Boolean	boolean, java.lang.Boolean
NumericBooleanType	INTEGER, 0 is false, 1 is true	boolean, java.lang.Boolean	numeric_boolean

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
YesNoType	CHAR, 'N'/'n' is false, 'Y'/'y' is true. The uppercase value is written to the database.	boolean, java.lang.Boolean	yes_no
TrueFalseType	CHAR, 'F'/'f' is false, 'T'/'t' is true. The uppercase value is written to the database.	boolean, java.lang.Boolean	true_false
ByteType	TINYINT	byte, java.lang.Byte	byte, java.lang.Byte
ShortType	SMALLINT	short, java.lang.Short	short, java.lang.Short
IntegerTypes	INTEGER	int, java.lang.Integer	int, java.lang.Integer
LongType	BIGINT	long, java.lang.Long	long, java.lang.Long
FloatType	FLOAT	float, java.lang.Float	float, java.lang.Float
DoubleType	DOUBLE	double, java.lang.Double	double, java.lang.Double
BigIntegerType	NUMERIC	java.math.BigInteger	big_integer, java.math.BigInteger
BigDecimalType	NUMERIC	java.math.BigDecimal	big_decimal, java.math.bigDecimal
TimestampType	TIMESTAMP	java.sql.Timestamp	timestamp, java.sql.Timestamp
TimeType	TIME	java.sql.Time	time, java.sql.Time
DateType	DATE	java.sql.Date	date, java.sql.Date
CalendarType	TIMESTAMP	java.util.Calendar	calendar, java.util.Calendar
CalendarDateType	DATE	java.util.Calendar	calendar_date
CalendarTimeType	TIME	java.util.Calendar	calendar_time
CurrencyType	java.util.Currency	VARCHAR	currency, java.util.Currency

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
LocaleType	VARCHAR	java.util.Locale	locale, java.utility.locale
TimeZoneType	VARCHAR, using the TimeZone ID	java.util.TimeZone	timezone, java.util.TimeZone
UrlType	VARCHAR	java.net.URL	url, java.net.URL
ClassType	VARCHAR (class FQN)	java.lang.Class	class, java.lang.Class
BlobType	BLOB	java.sql.Blob	blob, java.sql.Blob
ClobType	CLOB	java.sql.Clob	clob, java.sql.Clob
BinaryType	VARBINARY	byte[]	binary, byte[]
MaterializedBlobType	BLOB	byte[]	materized_blob
ImageType	LONGVARBINARY	byte[]	image
WrapperBinaryType	VARBINARY	java.lang.Byte[]	wrapper-binary, Byte[], java.lang.Byte[]
CharArrayType	VARCHAR	char[]	characters, char[]
CharacterArrayType	VARCHAR	java.lang.Character[]	wrapper-characters, Character[], java.lang.Character[]
UUIDBinaryType	BINARY	java.util.UUID	uuid-binary, java.util.UUID
UUIDCharType	CHAR, can also read VARCHAR	java.util.UUID	uuid-char
PostgresUUIDType	PostgreSQL UUID, through Types#OTHER, which complies to the PostgreSQL JDBC driver definition	java.util.UUID	pg-uuid

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
SerializableType	VARBINARY	implementors of java.lang.Serializable	Unlike the other value types, multiple instances of this type are registered. It is registered once under java.io.Serializable, and registered under the specific java.io.Serializable implementation class names.
StringNVarcharType	NVARCHAR	java.lang.String	nstring
NTextType	LONGNVARCHAR	java.lang.String	ntext
NClobType	NCLOB	java.sql.NClob	nclob, java.sql.NClob
MaterializedNClobType	NCLOB	java.lang.String	materialized_nclob
PrimitiveCharacterArrayNClobType	NCHAR	char[]	N/A
CharacterNCharType	NCHAR	java.lang.Character	ncharacter
CharacterArrayNClobType	NCLOB	java.lang.Character[]	N/A

Table 2. Java 8 BasicTypes

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
DurationType	BIGINT	java.time.Duration	Duration, java.time.Duration
InstantType	TIMESTAMP	java.time.Instant	Instant, java.time.Instant
LocalDateTimeType	TIMESTAMP	java.time.LocalDateTime	LocalDateTime, java.time.LocalDateTime
LocalDateType	DATE	java.time.LocalDate	LocalDate, java.time.LocalDate
LocalTimeType	TIME	java.time.LocalTime	LocalTime, java.time.LocalTime
OffsetDateTimeType	TIMESTAMP	java.time.OffsetDateTime	OffsetDateTime, java.time.OffsetDateTime
OffsetTimeType	TIME	java.time.OffsetTime	OffsetTime, java.time.OffsetTime

Hibernate type (org.hibernate.type package)	JDBC type	Java type	BasicTypeRegistry key(s)
OffsetTimeType	TIMESTAMP	java.time.ZonedDateTime	ZonedDateTime, java.time.ZonedDateTime

Table 3. Hibernate Spatial BasicTypes

Hibernate type (org.hibernate.spatial package)	JDBC type	Java type	BasicTypeRegistry key(s)
JTSGeometryType	depends on the dialect	com.vividsolutions.jts.geom.Geometry	jts_geometry, or the classname of Geometry or any of its subclasses
GeolatteGeometryType	depends on the dialect	org.geolatte.geom.Geometry	geolatte_geometry, or the classname of Geometry or any of its subclasses

> To use these hibernate-spatial types, you must add the `hibernate-spatial` dependency to your classpath *and* use a `org.hibernate.spatial.SpatialDialect` implementation. See Spatial for more details about spatial types.

These mappings are managed by a service inside Hibernate called the `org.hibernate.type.BasicTypeRegistry`, which essentially maintains a map of `org.hibernate.type.BasicType` (a `org.hibernate.type.Type` specialization) instances keyed by a name. That is the purpose of the "BasicTypeRegistry key(s)" column in the previous tables.

2.3.2. The `@Basic` annotation

Strictly speaking, a basic type is denoted by the `javax.persistence.Basic` annotation. Generally speaking, the `@Basic` annotation can be ignored, as it is assumed by default. Both of the following examples are ultimately the same.

Example 3. `@Basic` declared explicitly

```java
@Entity(name = "Product")
public class Product {

    @Id
    @Basic
    private Integer id;

    @Basic
    private String sku;

    @Basic
    private String name;

    @Basic
    private String description;
}
```

Example 4. `@Basic` *being implicitly implied*

```java
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String sku;

    private String name;

    private String description;
}
```

The JPA specification strictly limits the Java types that can be marked as basic to the following listing:

- Java primitive types (`boolean` , `int` , etc)
- wrappers for the primitive types (`java.lang.Boolean` , `java.lang.Integer` , etc)

- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]` or `Byte[]`
- `char[]` or `Character[]`
- `enums`
- any other type that implements `Serializable` (JPA's "support" for `Serializable` types is to directly serialize their state to the database).

If provider portability is a concern, you should stick to just these basic types. Note that JPA 2.1 did add the notion of a `javax.persistence.AttributeConverter` to help alleviate some of these concerns; see JPA 2.1 AttributeConverters for more on this topic.

The `@Basic` annotation defines 2 attributes.

`optional` - boolean (defaults to true)
Defines whether this attribute allows nulls. JPA defines this as "a hint", which essentially means that it effect is specifically required. As long as the type is not primitive, Hibernate takes this to mean that the underlying column should be `NULLABLE` .

`fetch` - FetchType (defaults to EAGER)

Defines whether this attribute should be fetched eagerly or lazily. JPA says that EAGER is a requirement to the provider (Hibernate) that the value should be fetched when the owner is fetched, while LAZY is merely a hint that the value be fetched when the attribute is accessed. Hibernate ignores this setting for basic types unless you are using bytecode enhancement. See the BytecodeEnhancement for additional information on fetching and on bytecode enhancement.

2.3.3. The `@Column` annotation

JPA defines rules for implicitly determining the name of tables and columns. For a detailed discussion of implicit naming see Naming.

For basic type attributes, the implicit naming rule is that the column name is the same as the attribute name. If that implicit naming rule does not meet your requirements, you can explicitly tell Hibernate (and other providers) the column name to use.

Example 5. Explicit column naming

JAVA

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String sku;

    private String name;

    @Column( name = "NOTES" )
    private String description;
}
```

Here we use `@Column` to explicitly map the `description` attribute to the `NOTES` column, as opposed to the implicit column name `description`.

The `@Column` annotation defines other mapping information as well. See its Javadocs for details.

2.3.4. BasicTypeRegistry

We said before that a Hibernate type is not a Java type, nor a SQL type, but that it understands both and performs the marshalling between them. But looking at the basic type mappings from the previous examples, how did Hibernate know to use its `org.hibernate.type.StringType` for mapping for `java.lang.String` attributes, or its `org.hibernate.type.IntegerType` for mapping `java.lang.Integer` attributes?

The answer lies in a service inside Hibernate called the `org.hibernate.type.BasicTypeRegistry`, which essentially maintains a map of `org.hibernate.type.BasicType` (a `org.hibernate.type.Type` specialization) instances keyed by a name.

We will see later, in the Explicit BasicTypes section, that we can explicitly tell Hibernate which BasicType to use for a particular attribute. But first let's explore how implicit resolution works and how applications can adjust implicit resolution.

> A thorough discussion of the `BasicTypeRegistry` and all the different ways to contribute types to it is beyond the scope of this documentation. Please see Integrations Guide for complete details.

As an example, take a String attribute such as we saw before with Product#sku. Since there was no explicit type mapping, Hibernate looks to the `BasicTypeRegistry` to find the registered mapping for `java.lang.String`. This goes back to the "BasicTypeRegistry key(s)" column we saw in the tables at the start of this chapter.

As a baseline within `BasicTypeRegistry`, Hibernate follows the recommended mappings of JDBC for Java types. JDBC recommends mapping Strings to VARCHAR, which is the exact mapping that `StringType` handles. So that is the baseline mapping within `BasicTypeRegistry` for Strings.

Applications can also extend (add new `BasicType` registrations) or override (replace an existing `BasicType` registration) using one of the `MetadataBuilder#applyBasicType` methods or the `MetadataBuilder#applyTypes` method during bootstrap. For more details, see Custom BasicTypes section.

2.3.5. Explicit BasicTypes

Sometimes you want a particular attribute to be handled differently. Occasionally Hibernate will implicitly pick a `BasicType` that you do not want (and for some reason you do not want to adjust the `BasicTypeRegistry`).

In these cases you must explicitly tell Hibernate the `BasicType` to use, via the `org.hibernate.annotations.Type` annotation.

Example 6. Using `@org.hibernate.annotations.Type`

JAVA

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String sku;

    @org.hibernate.annotations.Type( type = "nstring" )
    private String name;

    @org.hibernate.annotations.Type( type = "materialized_nclob" )
    private String description;
}
```

This tells Hibernate to store the Strings as nationalized data. This is just for illustration purposes; for better ways to indicate nationalized character data see Mapping Nationalized Character Data section.

Additionally, the description is to be handled as a LOB. Again, for better ways to indicate LOBs see Mapping LOBs section.

The `org.hibernate.annotations.Type#type` attribute can name any of the following:

- Fully qualified name of any `org.hibernate.type.Type` implementation
- Any key registered with `BasicTypeRegistry`
- The name of any known *type definitions*

2.3.6. Custom BasicTypes

Hibernate makes it relatively easy for developers to create their own basic type mappings type. For example, you might want to persist properties of type `java.util.BigInteger` to `VARCHAR` columns, or support completely new types.

There are two approaches to developing a custom type:

- implementing a `BasicType` and registering it
- implementing a `UserType` which doesn't require type registration

As a means of illustrating the different approaches, let's consider a use case where we need to support a `java.util.BitSet` mapping that's stored as a VARCHAR.

Implementing a `BasicType`

The first approach is to directly implement the `BasicType` interface.

> Because the `BasicType` interface has a lot of methods to implement, it's much more convenient to extend the `AbstractStandardBasicType`, or the `AbstractSingleColumnStandardBasicType` if the value is stored in a single database column.

First, we need to extend the `AbstractSingleColumnStandardBasicType` like this:

Example 7. Custom `BasicType` implementation

JAVA

```
public class BitSetType
        extends AbstractSingleColumnStandardBasicType<BitSet>
        implements DiscriminatorType<BitSet> {

    public static final BitSetType INSTANCE = new BitSetType();

    public BitSetType() {
        super( VarcharTypeDescriptor.INSTANCE, BitSetTypeDescriptor.INSTANCE );
    }

    @Override
    public BitSet stringToObject(String xml) throws Exception {
        return fromString( xml );
    }

    @Override
    public String objectToSQLString(BitSet value, Dialect dialect) throws Exception {
        return toString( value );
    }

    @Override
    public String getName() {
        return "bitset";
    }

}
```

The `AbstractSingleColumnStandardBasicType` requires an `sqlTypeDescriptor` and a `javaTypeDescriptor`. The `sqlTypeDescriptor` is `VarcharTypeDescriptor.INSTANCE` because the database column is a VARCHAR. On the Java side, we need to use a `BitSetTypeDescriptor` instance which can be implemented like this:

Example 8. Custom `AbstractTypeDescriptor` implementation

JAVA

```
public class BitSetTypeDescriptor extends AbstractTypeDescriptor<BitSet> {

    private static final String DELIMITER = ",";

    public static final BitSetTypeDescriptor INSTANCE = new BitSetTypeDescriptor();

    public BitSetTypeDescriptor() {
        super( BitSet.class );
    }

    @Override
    public String toString(BitSet value) {
        StringBuilder builder = new StringBuilder();
        for ( long token : value.toLongArray() ) {
            if ( builder.length() > 0 ) {
                builder.append( DELIMITER );
            }
            builder.append( Long.toString( token, 2 ) );
        }
        return builder.toString();
    }

    @Override
    public BitSet fromString(String string) {
        if ( string == null || string.isEmpty() ) {
            return null;
        }
        String[] tokens = string.split( DELIMITER );
        long[] values = new long[tokens.length];

        for ( int i = 0; i < tokens.length; i++ ) {
            values[i] = Long.valueOf( tokens[i], 2 );
        }
        return BitSet.valueOf( values );
    }

    @SuppressWarnings({"unchecked"})
    public <X> X unwrap(BitSet value, Class<X> type, WrapperOptions options) {
        if ( value == null ) {
            return null;
        }
        if ( BitSet.class.isAssignableFrom( type ) ) {
            return (X) value;
        }
        if ( String.class.isAssignableFrom( type ) ) {
            return (X) toString( value);
        }
        throw unknownUnwrap( type );
    }

    public <X> BitSet wrap(X value, WrapperOptions options) {
        if ( value == null ) {
            return null;
        }
        if ( String.class.isInstance( value ) ) {
            return fromString( (String) value );
```

```
        }
        if ( BitSet.class.isInstance( value ) ) {
            return (BitSet) value;
        }
        throw unknownWrap( value.getClass() );
    }
}
```

The `unwrap` method is used when passing a `BitSet` as a `PreparedStatement` bind parameter, while the `wrap` method is used to transform the JDBC column value object (e.g. `String` in our case) to the actual mapping object type (e.g. `BitSet` in this example).

The `BasicType` must be registered, and this can be done at bootstrapping time:

Example 9. Register a Custom `BasicType` implementation

JAVA

```
configuration.registerTypeContributor( (typeContributions, serviceRegistry) -> {
    typeContributions.contributeType( BitSetType.INSTANCE );
} );
```

or using the `MetadataBuilder`

JAVA

```
ServiceRegistry standardRegistry =
    new StandardServiceRegistryBuilder().build();

MetadataSources sources = new MetadataSources( standardRegistry );

MetadataBuilder metadataBuilder = sources.getMetadataBuilder();

metadataBuilder.applyBasicType( BitSetType.INSTANCE );
```

With the new `BitSetType` being registered as `bitset`, the entity mapping looks like this:

Example 10. Custom `BasicType` mapping

```java
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    @Type( type = "bitset" )
    private BitSet bitSet;

    public Integer getId() {
        return id;
    }

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

    public BitSet getBitSet() {
        return bitSet;
    }

    public void setBitSet(BitSet bitSet) {
        this.bitSet = bitSet;
    }
}
```

To validate this new `BasicType` implementation, we can test it as follows:

Example 11. Persisting the custom `BasicType`

```java
BitSet bitSet = BitSet.valueOf( new long[] {1, 2, 3} );

doInHibernate( this::sessionFactory, session -> {
    Product product = new Product( );
    product.setId( 1 );
    product.setBitSet( bitSet );
    session.persist( product );
} );

doInHibernate( this::sessionFactory, session -> {
    Product product = session.get( Product.class, 1 );
    assertEquals(bitSet, product.getBitSet());
} );
```

When executing this unit test, Hibernate generates the following SQL statements:

Example 12. Persisting the custom `BasicType`

```
DEBUG SQL:92 -
    insert
    into
        Product
        (bitSet, id)
    values
        (?, ?)

TRACE BasicBinder:65 - binding parameter [1] as [VARCHAR] - [{0, 65, 128, 129}]
TRACE BasicBinder:65 - binding parameter [2] as [INTEGER] - [1]

DEBUG SQL:92 -
    select
        bitsettype0_.id as id1_0_0_,
        bitsettype0_.bitSet as bitSet2_0_0_
    from
        Product bitsettype0_
    where
        bitsettype0_.id=?

TRACE BasicBinder:65 - binding parameter [1] as [INTEGER] - [1]
TRACE BasicExtractor:61 - extracted value ([bitSet2_0_0_] : [VARCHAR]) - [{0, 65, 128, 129}]
```

JAVA

As you can see, the `BitSetType` takes care of the *Java-to-SQL* and *SQL-to-Java* type conversion.

Implementing a `UserType`

The second approach is to implement the `UserType` interface.

Example 13. Custom `UserType` implementation

JAVA

```
public class BitSetUserType implements UserType {

        public static final BitSetUserType INSTANCE = new BitSetUserType();

    private static final Logger log = Logger.getLogger( BitSetUserType.class );

    @Override
    public int[] sqlTypes() {
        return new int[] {StringType.INSTANCE.sqlType()};
    }

    @Override
    public Class returnedClass() {
        return String.class;
    }

    @Override
    public boolean equals(Object x, Object y)
                        throws HibernateException {
        return Objects.equals( x, y );
    }

    @Override
    public int hashCode(Object x)
                        throws HibernateException {
        return Objects.hashCode( x );
    }

    @Override
    public Object nullSafeGet(
            ResultSet rs, String[] names, SharedSessionContractImplementor session, Object owner)
            throws HibernateException, SQLException {
        String columnName = names[0];
        String columnValue = (String) rs.getObject( columnName );
        log.debugv("Result set column {0} value is {1}", columnName, columnValue);
        return columnValue == null ? null :
                               BitSetTypeDescriptor.INSTANCE.fromString( columnValue );
    }

    @Override
    public void nullSafeSet(
            PreparedStatement st, Object value, int index, SharedSessionContractImplementor session)
            throws HibernateException, SQLException {
        if ( value == null ) {
            log.debugv("Binding null to parameter {0} ",index);
            st.setNull( index, Types.VARCHAR );
        }
        else {
            String stringValue = BitSetTypeDescriptor.INSTANCE.toString( (BitSet) value );
            log.debugv("Binding {0} to parameter {1} ", stringValue, index);
            st.setString( index, stringValue );
        }
    }

    @Override
    public Object deepCopy(Object value)
```

```
                        throws HibernateException {
            return value == null ? null :
                BitSet.valueOf( BitSet.class.cast( value ).toLongArray() );
        }

        @Override
        public boolean isMutable() {
            return true;
        }

        @Override
        public Serializable disassemble(Object value)
                        throws HibernateException {
            return (BitSet) deepCopy( value );
        }

        @Override
        public Object assemble(Serializable cached, Object owner)
                        throws HibernateException {
            return deepCopy( cached );
        }

        @Override
        public Object replace(Object original, Object target, Object owner)
                        throws HibernateException {
            return deepCopy( original );
        }
    }
```

The entity mapping looks as follows:

Example 14. Custom `UserType` mapping

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    @Type( type = "bitset" )
    private BitSet bitSet;

    public Integer getId() {
        return id;
    }

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

    public BitSet getBitSet() {
        return bitSet;
    }

    public void setBitSet(BitSet bitSet) {
        this.bitSet = bitSet;
    }
}
```
JAVA

In this example, the `UserType` is registered under the `bitset` name, and this is done like this:

Example 15. Register a Custom `UserType` implementation

```
configuration.registerTypeContributor( (typeContributions, serviceRegistry) -> {
    typeContributions.contributeType( BitSetUserType.INSTANCE, "bitset");
} );
```
JAVA

or using the `MetadataBuilder`

```
ServiceRegistry standardRegistry =
    new StandardServiceRegistryBuilder().build();

MetadataSources sources = new MetadataSources( standardRegistry );

MetadataBuilder metadataBuilder = sources.getMetadataBuilder();

metadataBuilder.applyBasicType( BitSetUserType.INSTANCE, "bitset" );
```
JAVA

Like `BasicType`, you can also register the `UserType` using a simple name.

Without registration, the `UserType` mapping requires the fully-classified name:

```
@Type( type = "org.hibernate.userguide.mapping.basic.BitSetUserType" )
```
JAVA

When running the previous test case against the `BitSetUserType` entity mapping, Hibernate executed the following SQL statements:

Example 16. Persisting the custom `BasicType`

```
DEBUG SQL:92 -
    insert
    into
        Product
        (bitSet, id)
    values
        (?, ?)

DEBUG BitSetUserType:71 - Binding 1,10,11 to parameter 1
TRACE BasicBinder:65 - binding parameter [2] as [INTEGER] - [1]

DEBUG SQL:92 -
    select
        bitsetuser0_.id as id1_0_0_,
        bitsetuser0_.bitSet as bitSet2_0_0_
    from
        Product bitsetuser0_
    where
        bitsetuser0_.id=?

TRACE BasicBinder:65 - binding parameter [1] as [INTEGER] - [1]
DEBUG BitSetUserType:56 - Result set column bitSet2_0_0_ value is 1,10,11
```
JAVA

2.3.7. Mapping enums

Hibernate supports the mapping of Java enums as basic value types in a number of different ways.

`@Enumerated`

The original JPA-compliant way to map enums was via the `@Enumerated` and `@MapKeyEnumerated` for map keys annotations which works on the principle that the enum values are stored according to one of 2 strategies indicated by `javax.persistence.EnumType`:

`ORDINAL`
: stored according to the enum value's ordinal position within the enum class, as indicated by `java.lang.Enum#ordinal`

`STRING`
: stored according to the enum value's name, as indicated by `java.lang.Enum#name`

Assuming the following enumeration:

Example 17. `PhoneType` *enumeration*

```
public enum PhoneType {
    LAND_LINE,
    MOBILE;
}
```
JAVA

In the ORDINAL example, the `phone_type` column is defined as an (nullable) INTEGER type and would hold:

`NULL`
: For null values

`0`
: For the `LAND_LINE` enum

`1`
: For the `MOBILE` enum

Example 18. `@Enumerated(ORDINAL)` *example*

```java
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    @Column(name = "phone_number")
    private String number;

    @Enumerated(EnumType.ORDINAL)
    @Column(name = "phone_type")
    private PhoneType type;

    //Getters and setters are omitted for brevity

}
```

When persisting this entity, Hibernate generates the following SQL statement:

Example 19. Persisting an entity with an `@Enumerated(ORDINAL)` mapping

```java
Phone phone = new Phone( );
phone.setId( 1L );
phone.setNumber( "123-456-78990" );
phone.setType( PhoneType.MOBILE );
entityManager.persist( phone );
```

```sql
INSERT INTO Phone (phone_number, phone_type, id)
VALUES ('123-456-78990', 2, 1)
```

In the STRING example, the `phone_type` column is defined as a (nullable) VARCHAR type and would hold:

`NULL`
: For null values

`LAND_LINE`
: For the `LAND_LINE` enum

`MOBILE`
: For the `MOBILE` enum

Example 20. `@Enumerated(STRING)` example

```java
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    @Column(name = "phone_number")
    private String number;

    @Enumerated(EnumType.STRING)
    @Column(name = "phone_type")
    private PhoneType type;

    //Getters and setters are omitted for brevity

}
```

Persisting the same entity like in the `@Enumerated(ORDINAL)` example, Hibernate generates the following SQL statement:

Example 21. Persisting an entity with an `@Enumerated(STRING)` mapping

```sql
INSERT INTO Phone (phone_number, phone_type, id)
VALUES ('123-456-78990', 'MOBILE', 1)
```

AttributeConverter

Let's consider the following `Gender` enum which stores its values using the `'M'` and `'F'` codes.

Example 22. Enum with custom constructor

```java
public enum Gender {

    MALE( 'M' ),
    FEMALE( 'F' );

    private final char code;

    Gender(char code) {
        this.code = code;
    }

    public static Gender fromCode(char code) {
        if ( code == 'M' || code == 'm' ) {
            return MALE;
        }
        if ( code == 'F' || code == 'f' ) {
            return FEMALE;
        }
        throw new UnsupportedOperationException(
            "The code " + code + " is not supported!"
        );
    }

    public char getCode() {
        return code;
    }
}
```

You can map enums in a JPA compliant way using a JPA 2.1 AttributeConverter.

Example 23. Enum mapping with `AttributeConverter` example

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String name;

    @Convert( converter = GenderConverter.class )
    public Gender gender;

    //Getters and setters are omitted for brevity

}

@Converter
public static class GenderConverter
        implements AttributeConverter<Gender, Character> {

    public Character convertToDatabaseColumn( Gender value ) {
        if ( value == null ) {
            return null;
        }

        return value.getCode();
    }

    public Gender convertToEntityAttribute( Character value ) {
        if ( value == null ) {
            return null;
        }

        return Gender.fromCode( value );
    }
}
```

Here, the gender column is defined as a CHAR type and would hold:

`NULL`
: For null values

`'M'`
: For the `MALE` enum

`'F'`
: For the `FEMALE` enum

For additional details on using AttributeConverters, see JPA 2.1 AttributeConverters section.

> JPA explicitly disallows the use of an AttributeConverter with an attribute marked as `@Enumerated`. So if using the AttributeConverter approach, be sure not to mark the attribute as `@Enumerated`.

Custom type

You can also map enums using a Hibernate custom type mapping. Let's again revisit the Gender enum example, this time using a custom Type to store the more standardized `'M'` and `'F'` codes.

Example 24. Enum mapping with custom Type example

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String name;

    @Type( type = "org.hibernate.userguide.mapping.basic.GenderType" )
    public Gender gender;

    //Getters and setters are omitted for brevity

}

public class GenderType extends AbstractSingleColumnStandardBasicType<Gender> {

    public static final GenderType INSTANCE = new GenderType();

    public GenderType() {
        super(
            CharTypeDescriptor.INSTANCE,
            GenderJavaTypeDescriptor.INSTANCE
        );
    }

    public String getName() {
        return "gender";
    }

    @Override
    protected boolean registerUnderJavaType() {
        return true;
    }
}

public class GenderJavaTypeDescriptor extends AbstractTypeDescriptor<Gender> {

    public static final GenderJavaTypeDescriptor INSTANCE =
        new GenderJavaTypeDescriptor();

    protected GenderJavaTypeDescriptor() {
        super( Gender.class );
    }

    public String toString(Gender value) {
        return value == null ? null : value.name();
    }

    public Gender fromString(String string) {
        return string == null ? null : Gender.valueOf( string );
    }

    public <X> X unwrap(Gender value, Class<X> type, WrapperOptions options) {
        return CharacterTypeDescriptor.INSTANCE.unwrap(
            value == null ? null : value.getCode(),
```

```
                type,
                options
            );
        }

        public <X> Gender wrap(X value, WrapperOptions options) {
            return Gender.fromCode(
                CharacterTypeDescriptor.INSTANCE.wrap( value, options )
            );
        }
    }
```

Again, the gender column is defined as a CHAR type and would hold:

`NULL`
: For null values

`'M'`
: For the `MALE` enum

`'F'`
: For the `FEMALE` enum

For additional details on using custom types, see Custom BasicTypes section.

2.3.8. Mapping LOBs

Mapping LOBs (database Large Objects) come in 2 forms, those using the JDBC locator types and those materializing the LOB data.

JDBC LOB locators exist to allow efficient access to the LOB data. They allow the JDBC driver to stream parts of the LOB data as needed, potentially freeing up memory space. However they can be unnatural to deal with and have certain limitations. For example, a LOB locator is only portably valid during the duration of the transaction in which it was obtained.

The idea of materialized LOBs is to trade-off the potential efficiency (not all drivers handle LOB data efficiently) for a more natural programming paradigm using familiar Java types such as String or byte[], etc for these LOBs.

Materialized deals with the entire LOB contents in memory, whereas LOB locators (in theory) allow streaming parts of the LOB contents into memory as needed.

The JDBC LOB locator types include:

- `java.sql.Blob`
- `java.sql.Clob`
- `java.sql.NClob`

Mapping materialized forms of these LOB values would use more familiar Java types such as `String`, `char[]`, `byte[]`, etc. The trade off for *more familiar* is usually performance.

For a first look, let's assume we have a `CLOB` column that we would like to map (`NCLOB` character `LOB` data will be covered in Mapping Nationalized Character Data section).

Example 25. CLOB - SQL

```sql
CREATE TABLE Product (
  id INTEGER NOT NULL
  image clob
  name VARCHAR(255)
  PRIMARY KEY ( id )
)
```

Let's first map this using the `@Lob` JPA annotation and the `java.sql.Clob` type:

Example 26. `CLOB` mapped to `java.sql.Clob`

```java
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    private Clob warranty;

    //Getters and setters are omitted for brevity

}
```

To persist such an entity, you have to create a `Clob` using plain JDBC:

Example 27. Persisting a `java.sql.Clob` entity

```
String warranty = "My product warranty";

final Product product = new Product();
product.setId( 1 );
product.setName( "Mobile phone" );

session.doWork( connection -> {
    product.setWarranty( ClobProxy.generateProxy( warranty ) );
} );

entityManager.persist( product );
```

JAVA

To retrieve the `Clob` content, you need to transform the underlying `java.io.Reader` :

Example 28. Returning a `java.sql.Clob` entity

```
Product product = entityManager.find( Product.class, productId );
try (Reader reader = product.getWarranty().getCharacterStream()) {
    assertEquals( "My product warranty", toString( reader ) );
}
```

JAVA

We could also map the CLOB in a materialized form. This way, we can either use a `String` or a `char[]`.

Example 29. `CLOB` mapped to `String`

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    private String warranty;

    //Getters and setters are omitted for brevity

}
```

JAVA

> How JDBC deals with `LOB` data varies from driver to driver, and Hibernate tries to handle all these variances on your behalf.
>
> However, some drivers are trickier (e.g. PostgreSQL JDBC drivers), and, in such cases, you may have to do some extra to get LOBs working. Such discussions are beyond the scope of this guide.

We might even want the materialized data as a char array (for some crazy reason).

Example 30. CLOB - materialized `char[]` mapping

```
JAVA
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    private char[] warranty;

    //Getters and setters are omitted for brevity

}
```

`BLOB` data is mapped in a similar fashion.

Example 31. BLOB - SQL

```
SQL
CREATE TABLE Product (
    id INTEGER NOT NULL ,
    image blob ,
    name VARCHAR(255) ,
    PRIMARY KEY ( id )
)
```

Let's first map this using the JDBC `java.sql.Blob` type.

Example 32. `BLOB` *mapped to* `java.sql.Blob`

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    private Blob image;

    //Getters and setters are omitted for brevity

}
```

JAVA

To persist such an entity, you have to create a `Blob` using plain JDBC:

Example 33. Persisting a `java.sql.Blob` *entity*

```
byte[] image = new byte[] {1, 2, 3};

final Product product = new Product();
product.setId( 1 );
product.setName( "Mobile phone" );

session.doWork( connection -> {
    product.setImage( BlobProxy.generateProxy( image ) );
} );

entityManager.persist( product );
```

JAVA

To retrieve the `Blob` content, you need to transform the underlying `java.io.Reader` :

Example 34. Returning a `java.sql.Blob` *entity*

```
Product product = entityManager.find( Product.class, productId );

try (InputStream inputStream = product.getImage().getBinaryStream()) {
    assertArrayEquals(new byte[] {1, 2, 3}, toBytes( inputStream ) );
}
```

JAVA

We could also map the BLOB in a materialized form (e.g. `byte[]`).

Example 35. `BLOB` *mapped to* `byte[]`

```
JAVA
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    private byte[] image;

    //Getters and setters are omitted for brevity

}
```

2.3.9. Mapping Nationalized Character Data

JDBC 4 added the ability to explicitly handle nationalized character data. To this end it added specific nationalized character data types.

- NCHAR
- NVARCHAR
- LONGNVARCHAR
- NCLOB

Example 36. `NVARCHAR` *- SQL*

```
JAVA
CREATE TABLE Product (
    id INTEGER NOT NULL ,
    name VARCHAR(255) ,
    warranty NVARCHAR(255) ,
    PRIMARY KEY ( id )
)
```

To map a specific attribute to a nationalized variant data type, Hibernate defines the `@Nationalized` annotation.

Example 37. `NVARCHAR` *mapping*

```java
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Nationalized
    private String warranty;

    //Getters and setters are omitted for brevity

}
```

Just like with `CLOB`, Hibernate can also deal with `NCLOB` SQL data types:

Example 38. `NCLOB` *- SQL*

```java
CREATE TABLE Product (
    id INTEGER NOT NULL ,
    name VARCHAR(255) ,
    warranty nclob ,
    PRIMARY KEY ( id )
)
```

Hibernate can map the `NCLOB` to a `java.sql.NClob`

Example 39. `NCLOB` *mapped to* `java.sql.NClob`

```java
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    @Nationalized
    // Clob also works, because NClob extends Clob.
    // The database type is still NCLOB either way and handled as such.
    private NClob warranty;

    //Getters and setters are omitted for brevity

}
```

To persist such an entity, you have to create a `NClob` using plain JDBC:

Example 40. Persisting a `java.sql.NClob` entity

```java
String warranty = "My product warranty";

final Product product = new Product();
product.setId( 1 );
product.setName( "Mobile phone" );

session.doWork( connection -> {
    product.setWarranty( connection.createNClob() );
    product.getWarranty().setString( 1, warranty );
} );

entityManager.persist( product );
```

To retrieve the `NClob` content, you need to transform the underlying `java.io.Reader`:

Example 41. Returning a `java.sql.NClob` entity

```java
Product product = entityManager.find( Product.class, productId );
try (Reader reader = product.getWarranty().getCharacterStream()) {
    assertEquals( "My product warranty", toString( reader ) );
}
```

We could also map the `NCLOB` in a materialized form. This way, we can either use a `String` or a `char[]`.

Example 42. `NCLOB` *mapped to* `String`

JAVA

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    @Nationalized
    private String warranty;

    //Getters and setters are omitted for brevity

}
```

We might even want the materialized data as a char array.

Example 43. NCLOB - materialized `char[]` *mapping*

JAVA

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    private String name;

    @Lob
    @Nationalized
    private char[] warranty;

    //Getters and setters are omitted for brevity

}
```

> If you application and database are entirely nationalized you may instead want to enable nationalized character data as the default. You can do this via the `hibernate.use_nationalized_character_data` setting or by calling `MetadataBuilder#enableGlobalNationalizedCharacterDataSupport` during bootstrap.

2.3.10. Mapping UUID Values

Hibernate also allows you to map UUID values, again in a number of ways.

> The default UUID mapping is as binary because it represents more efficient storage. However many applications prefer the readability of character storage. To switch the default mapping, simply call `MetadataBuilder.applyBasicType( UUIDCharType.INSTANCE, UUID.class.getName() )`.

2.3.11. UUID as binary

As mentioned, the default mapping for UUID attributes. Maps the UUID to a `byte[]` using `java.util.UUID#getMostSignificantBits` and `java.util.UUID#getLeastSignificantBits` and stores that as `BINARY` data.

Chosen as the default simply because it is generally more efficient from storage perspective.

2.3.12. UUID as (var)char

Maps the UUID to a String using `java.util.UUID#toString` and `java.util.UUID#fromString` and stores that as `CHAR` or `VARCHAR` data.

2.3.13. PostgeSQL-specific UUID

> When using one of the PostgreSQL Dialects, this becomes the default UUID mapping.

Maps the UUID using PostgreSQL's specific UUID data type. The PostgreSQL JDBC driver chooses to map its UUID type to the `OTHER` code. Note that this can cause difficulty as the driver chooses to map many different data types to `OTHER`.

2.3.14. UUID as identifier

Hibernate supports using UUID values as identifiers, and they can even be generated on user's behalf. For details, see the discussion of generators in *Identifier generators*.

2.3.15. Mapping Date/Time Values

Hibernate allows various Java Date/Time classes to be mapped as persistent domain model entity properties. The SQL standard defines three Date/Time types:

DATE
: Represents a calendar date by storing years, months and days. The JDBC equivalent is `java.sql.Date`

TIME
: Represents the time of a day and it stores hours, minutes and seconds. The JDBC equivalent is `java.sql.Time`

TIMESTAMP
: It stores both a DATE and a TIME plus nanoseconds. The JDBC equivalent is `java.sql.Timestamp`

> To avoid dependencies on the `java.sql` package, it's common to use the `java.util` or `java.time` Date/Time classes instead.

While the `java.sql` classes define a direct association to the SQL Date/Time data types, the `java.util` or `java.time` properties need to explicitly mark the SQL type correlation with the `@Temporal` annotation. This way, a `java.util.Date` or a `java.util.Calendar` can be mapped to either an SQL `DATE`, `TIME` or `TIMESTAMP` type.

Considering the following entity:

Example 44. `java.util.Date` *mapped as* `DATE`

```java
@Entity(name = "DateEvent")
public static class DateEvent {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`timestamp`")
    @Temporal(TemporalType.DATE)
    private Date timestamp;

    //Getters and setters are omitted for brevity

}
```

When persisting such entity:

Example 45. Persisting a `java.util.Date` mapping

```java
DateEvent dateEvent = new DateEvent( new Date() );
entityManager.persist( dateEvent );
```

Hibernate generates the following INSERT statement:

```sql
INSERT INTO DateEvent ( timestamp, id )
VALUES ( '2015-12-29', 1 )
```

Only the year, month and the day field were saved into the database.

If we change the `@Temporal` type to `TIME` :

Example 46. `java.util.Date` mapped as `TIME`

```java
@Column(name = "`timestamp`")
@Temporal(TemporalType.TIME)
private Date timestamp;
```

Hibernate will issue an INSERT statement containing the hour, minutes and seconds.

```
INSERT INTO DateEvent ( timestamp, id )
VALUES ( '16:51:58', 1 )
```
SQL

When the `@Temporal` type is set to `TIMESTAMP`:

Example 47. `java.util.Date` *mapped as* `TIMESTAMP`

```
@Column(name = "`timestamp`")
@Temporal(TemporalType.TIMESTAMP)
private Date timestamp;
```
JAVA

Hibernate will include both the `DATE`, the `TIME` and the nanoseconds in the INSERT statement:

```
INSERT INTO DateEvent ( timestamp, id )
VALUES ( '2015-12-29 16:54:04.544', 1 )
```
SQL

> Just like the `java.util.Date`, the `java.util.Calendar` requires the `@Temporal` annotation in order to know what JDBC data type to be chosen: DATE, TIME or TIMESTAMP. If the `java.util.Date` marks a point in time, the `java.util.Calendar` takes into consideration the default Time Zone.

Mapping Java 8 Date/Time Values

Java 8 came with a new Date/Time API, offering support for instant dates, intervals, local and zoned Date/Time immutable instances, bundled in the `java.time` package.

The mapping between the standard SQL Date/Time types and the supported Java 8 Date/Time class types looks as follows;

DATE
: `java.time.LocalDate`

TIME

`java.time.LocalTime`, `java.time.OffsetTime`

TIMESTAMP

`java.time.Instant`, `java.time.LocalDateTime`, `java.time.OffsetDateTime` and `java.time.ZonedDateTime`

> Because the mapping between Java 8 Date/Time classes and the SQL types is implicit, there is not need to specify the `@Temporal` annotation. Setting it on the `java.time` classes throws the following exception:
>
> ```
> org.hibernate.AnnotationException: @Temporal should only be set on a java.util.Date or
> java.util.Calendar property
> ```

Using a specific time zone

By default, Hibernate is going to use the `PreparedStatement.setTimestamp(int parameterIndex, java.sql.Timestamp)` (https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#setTimestamp-int-java.sql.Timestamp-) or `PreparedStatement.setTime(int parameterIndex, java.sql.Time x)` (https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#setTime-int-java.sql.Time-) when saving a `java.sql.Timestamp` or a `java.sql.Time` property.

When the time zone is not specified, the JDBC driver is going to use the underlying JVM default time zone, which might not be suitable if the application is used from all across the globe. For this reason, it is very common to use a single reference time zone (e.g. UTC) whenever saving/loading data from the database.

One alternative would be to configure all JVMs to use the reference time zone:

Declaratively

```
java -Duser.timezone=UTC ...
```
JAVA

Programmatically

```
TimeZone.setDefault( TimeZone.getTimeZone( "UTC" ) );
```
JAVA

However, as explained in this article (http://in.relation.to/2016/09/12/jdbc-time-zone-configuration-property/), this is not always practical especially for front-end nodes. For this reason, Hibernate offers the `hibernate.jdbc.time_zone` configuration property which can be configured:

Declaratively, at the `SessionFactory` level

JAVA

```
settings.put(
    AvailableSettings.JDBC_TIME_ZONE,
    TimeZone.getTimeZone( "UTC" )
);
```

Programmatically, on a per `Session` basis

JAVA

```
Session session = sessionFactory()
    .withOptions()
    .jdbcTimeZone( TimeZone.getTimeZone( "UTC" ) )
    .openSession();
```

With this configuration property in place, Hibernate is going to call the `PreparedStatement.setTimestamp(int parameterIndex, java.sql.Timestamp, Calendar cal)` (https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#setTimestamp-int-java.sql.Timestamp-java.util.Calendar-) or `PreparedStatement.setTime(int parameterIndex, java.sql.Time x, Calendar cal)` (https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#setTime-int-java.sql.Time-java.util.Calendar-), where the `java.util.Calendar` references the time zone provided via the `hibernate.jdbc.time_zone` property.

2.3.16. JPA 2.1 AttributeConverters

Although Hibernate has long been offering custom types, as a JPA 2.1 provider, it also supports `AttributeConverter` as well.

With a custom `AttributeConverter`, the application developer can map a given JDBC type to an entity basic type.

In the following example, the `java.util.Period` is going to be mapped to a `VARCHAR` database column.

Example 48. `java.util.Period` *custom* `AttributeConverter`

```java
@Converter
public class PeriodStringConverter
        implements AttributeConverter<Period, String> {

    @Override
    public String convertToDatabaseColumn(Period attribute) {
        return attribute.toString();
    }

    @Override
    public Period convertToEntityAttribute(String dbData) {
        return Period.parse( dbData );
    }
}
```

To make use of this custom converter, the `@Convert` annotation must decorate the entity attribute.

Example 49. Entity using the custom `java.util.Period AttributeConverter` mapping

```java
@Entity(name = "Event")
public static class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Convert(converter = PeriodStringConverter.class)
    @Column(columnDefinition = "")
    private Period span;

    //Getters and setters are omitted for brevity

}
```

When persisting such entity, Hibernate will do the type conversion based on the `AttributeConverter` logic:

Example 50. Persisting entity using the custom `AttributeConverter`

```sql
INSERT INTO Event ( span, id )
VALUES ( 'P1Y2M3D', 1 )
```

`AttributeConverter` Java and JDBC types

In cases when the Java type specified for the "database side" of the conversion (the second `AttributeConverter` bind parameter) is not known, Hibernate will fallback to a `java.io.Serializable` type.

If the Java type is not know to Hibernate, you will encounter the following message:

> *HHH000481: Encountered Java type for which we could not locate a JavaTypeDescriptor and which does not appear to implement equals and/or hashCode. This can lead to significant performance problems when performing equality/dirty checking involving this Java type. Consider registering a custom JavaTypeDescriptor or at least implementing equals/hashCode.*

Whether a Java type is "known" means it has an entry in the `JavaTypeDescriptorRegistry`. While by default Hibernate loads many JDK types into the `JavaTypeDescriptorRegistry`, an application can also expand the `JavaTypeDescriptorRegistry` by adding new `JavaTypeDescriptor` entries.

This way, Hibernate will also know how to handle a specific Java Object type at the JDBC level.

JPA 2.1 `AttributeConverter` Mutability Plan

A basic type that's converted by a JPA `AttributeConverter` is immutable if the underlying Java type is immutable and is mutable if the associated attribute type is mutable as well.

Therefore, mutability is given by the `JavaTypeDescriptor#getMutabilityPlan` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/type/descriptor/java/JavaTypeDescriptor.html#getMutabilityPlan--) of the associated entity attribute type.

Immutable types

If the entity attribute is a `String`, a primitive wrapper (e.g. `Integer`, `Long`) an Enum type, or any other immutable `Object` type, then you can only change the entity attribute value by reassigning it to a new value.

Considering we have the same `Period` entity attribute as illustrated in the JPA 2.1 AttributeConverters section:

JAVA

```
@Entity(name = "Event")
public static class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Convert(converter = PeriodStringConverter.class)
    @Column(columnDefinition = "")
    private Period span;

    //Getters and setters are omitted for brevity

}
```

The only way to change the `span` attribute is to reassign it to a different value:

JAVA

```
Event event = entityManager.createQuery( "from Event", Event.class ).getSingleResult();
event.setSpan(Period
    .ofYears( 3 )
    .plusMonths( 2 )
    .plusDays( 1 )
);
```

Mutable types

On the other hand, consider the following example where the `Money` type is a mutable.

JAVA

```
public static class Money {

    private long cents;

    public Money(long cents) {
        this.cents = cents;
    }

    public long getCents() {
        return cents;
    }

    public void setCents(long cents) {
        this.cents = cents;
    }
}

public static class MoneyConverter
        implements AttributeConverter<Money, Long> {

    @Override
    public Long convertToDatabaseColumn(Money attribute) {
        return attribute == null ? null : attribute.getCents();
    }

    @Override
    public Money convertToEntityAttribute(Long dbData) {
        return dbData == null ? null : new Money( dbData );
    }
}

@Entity(name = "Account")
public static class Account {

    @Id
    private Long id;

    private String owner;

    @Convert(converter = MoneyConverter.class)
    private Money balance;

    //Getters and setters are omitted for brevity

}
```

A mutable `Object` allows you to modify its internal structure, and Hibernate dirty checking mechanism is going to propagate the change to the database:

JAVA

```
Account account = entityManager.find( Account.class, 1L );
account.getBalance().setCents( 150 * 100L );
entityManager.persist( account );
```

> Although the `AttributeConverter` types can be mutable so that dirty checking, deep copying and second-level caching work properly, treating these as immutable (when they really are) is more efficient.
>
> For this reason, prefer immutable types over mutable ones whenever possible.

2.3.17. SQL quoted identifiers

You can force Hibernate to quote an identifier in the generated SQL by enclosing the table or column name in backticks in the mapping document. While traditionally, Hibernate used backticks for escaping SQL reserved keywords, JPA uses double quotes instead.

Once the reserved keywords are escaped, Hibernate will use the correct quotation style for the SQL `Dialect`. This is usually double quotes, but SQL Server uses brackets and MySQL uses backticks.

Example 51. Hibernate legacy quoting

JAVA

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity

}
```

Example 52. JPA quoting

```java
@Entity(name = "Product")
public static class Product {

    @Id
    private Long id;

    @Column(name = "\"name\"")
    private String name;

    @Column(name = "\"number\"")
    private String number;

    //Getters and setters are omitted for brevity

}
```

Because `name` and `number` are reserved words, the `Product` entity mapping uses backtricks to quote these column names.

When saving the following `Product entity` , Hibernate generates the following SQL insert statement:

Example 53. Persisting a quoted column name

```java
Product product = new Product();
product.setId( 1L );
product.setName( "Mobile phone" );
product.setNumber( "123-456-7890" );
entityManager.persist( product );
```

```sql
INSERT INTO Product ("name", "number", id)
VALUES ('Mobile phone', '123-456-7890', 1)
```

Global quoting

Hibernate can also quote all identifiers (e.g. table, columns) using the following configuration property:

```xml
<property
    name="hibernate.globally_quoted_identifiers"
    value="true"
/>
```

This way, we don't need to manually quote any identifier:

Example 54. JPA quoting

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Long id;

    private String name;

    private String number;

    //Getters and setters are omitted for brevity

}
```

JAVA

When persisting a `Product` entity, Hibernate is going to quote all identifiers as in the following example:

```
INSERT INTO "Product" ("name", "number", "id")
VALUES ('Mobile phone', '123-456-7890', 1)
```

SQL

As you can see, both the table name and all the column have been quoted.

For more about quoting-related configuration properties, check out the Mapping configurations section as well.

2.3.18. Generated properties

Generated properties are properties that have their values generated by the database. Typically, Hibernate applications needed to `refresh` objects that contain any properties for which the database was generating values. Marking properties as generated, however, lets the application delegate this responsibility to Hibernate. When Hibernate issues an SQL INSERT or UPDATE for an entity that has defined generated properties, it immediately issues a select to retrieve the generated values.

Properties marked as generated must additionally be *non-insertable* and *non-updateable*. Only `@Version` and `@Basic` types can be marked as generated.

`NEVER` (the default)
: the given property value is not generated within the database.

`INSERT`
: the given property value is generated on insert, but is not regenerated on subsequent updates. Properties like *creationTimestamp* fall into this category.

`ALWAYS`

the property value is generated both on insert and on update.

To mark a property as generated, use The Hibernate specific `@Generated` annotation.

`@Generated` annotation

The `@Generated` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Generated.html) annotation is used so that Hibernate can fetch the currently annotated property after the entity has been persisted or updated. For this reason, the `@Generated` annotation accepts a `GenerationTime` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenerationTime.html) enum value.

Considering the following entity:

Example 55. `@Generated` mapping example

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String middleName1;

    private String middleName2;

    private String middleName3;

    private String middleName4;

    private String middleName5;

    @Generated( value = GenerationTime.ALWAYS )
    @Column(columnDefinition =
        "AS CONCAT(" +
        "    COALESCE(firstName, ''), " +
        "    COALESCE(' ' + middleName1, ''), " +
        "    COALESCE(' ' + middleName2, ''), " +
        "    COALESCE(' ' + middleName3, ''), " +
        "    COALESCE(' ' + middleName4, ''), " +
        "    COALESCE(' ' + middleName5, ''), " +
        "    COALESCE(' ' + lastName, '') " +
        ")")
    private String fullName;

}
```

When the `Person` entity is persisted, Hibernate is going to fetch the calculated `fullName` column from the database, which concatenates the first, middle, and last name.

Example 56. `@Generated` *persist example*

JAVA

```
Person person = new Person();
person.setId( 1L );
person.setFirstName( "John" );
person.setMiddleName1( "Flávio" );
person.setMiddleName2( "André" );
person.setMiddleName3( "Frederico" );
person.setMiddleName4( "Rúben" );
person.setMiddleName5( "Artur" );
person.setLastName( "Doe" );

entityManager.persist( person );
entityManager.flush();

assertEquals("John Flávio André Frederico Rúben Artur Doe", person.getFullName());
```

```sql
INSERT INTO Person
(
    firstName,
    lastName,
    middleName1,
    middleName2,
    middleName3,
    middleName4,
    middleName5,
    id
)
values
(?, ?, ?, ?, ?, ?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [John]
-- binding parameter [2] as [VARCHAR] - [Doe]
-- binding parameter [3] as [VARCHAR] - [Flávio]
-- binding parameter [4] as [VARCHAR] - [André]
-- binding parameter [5] as [VARCHAR] - [Frederico]
-- binding parameter [6] as [VARCHAR] - [Rúben]
-- binding parameter [7] as [VARCHAR] - [Artur]
-- binding parameter [8] as [BIGINT]  - [1]

SELECT
    p.fullName as fullName3_0_
FROM
    Person p
WHERE
    p.id=?

-- binding parameter [1] as [BIGINT] - [1]
-- extracted value ([fullName3_0_] : [VARCHAR]) - [John Flávio André Frederico Rúben Artur Doe]
```

The same goes when the `Person` entity is updated. Hibernate is going to fetch the calculated `fullName` column from the database after the entity is modified.

Example 57. `@Generated` update example

```java
Person person = entityManager.find( Person.class, 1L );
person.setLastName( "Doe Jr" );

entityManager.flush();
assertEquals("John Flávio André Frederico Rúben Artur Doe Jr", person.getFullName());
```

```sql
UPDATE
    Person
SET
    firstName=?,
    lastName=?,
    middleName1=?,
    middleName2=?,
    middleName3=?,
    middleName4=?,
    middleName5=?
WHERE
    id=?

-- binding parameter [1] as [VARCHAR] - [John]
-- binding parameter [2] as [VARCHAR] - [Doe Jr]
-- binding parameter [3] as [VARCHAR] - [Flávio]
-- binding parameter [4] as [VARCHAR] - [André]
-- binding parameter [5] as [VARCHAR] - [Frederico]
-- binding parameter [6] as [VARCHAR] - [Rúben]
-- binding parameter [7] as [VARCHAR] - [Artur]
-- binding parameter [8] as [BIGINT]  - [1]

SELECT
    p.fullName as fullName3_0_
FROM
    Person p
WHERE
    p.id=?

-- binding parameter [1] as [BIGINT] - [1]
-- extracted value ([fullName3_0_] : [VARCHAR]) - [John Flávio André Frederico Rúben Artur Doe Jr]
```

`@GeneratorType` annotation

The `@GeneratorType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GeneratorType.html) annotation is used so that you can provide a custom generator to set the value of the currently annotated property.

For this reason, the `@GeneratorType` annotation accepts a `GenerationTime` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenerationTime.html) enum value and a custom `ValueGenerator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ValueGenerator.html) class type.

Considering the following entity:

Example 58. `@GeneratorType` *mapping example*

```java
public static class CurrentUser {

    public static final CurrentUser INSTANCE = new CurrentUser();

    private static final ThreadLocal<String> storage = new ThreadLocal<>();

    public void logIn(String user) {
        storage.set( user );
    }

    public void logOut() {
        storage.remove();
    }

    public String get() {
        return storage.get();
    }
}

public static class LoggedUserGenerator implements ValueGenerator<String> {

    @Override
    public String generateValue(
            Session session, Object owner) {
        return CurrentUser.INSTANCE.get();
    }
}

@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    @GeneratorType( type = LoggedUserGenerator.class, when = GenerationTime.INSERT)
    private String createdBy;

    @GeneratorType( type = LoggedUserGenerator.class, when = GenerationTime.ALWAYS)
    private String updatedBy;

}
```

When the `Person` entity is persisted, Hibernate is going to populate the `createdBy` column with the currently logged user.

Example 59. `@Generated` persist example

```
CurrentUser.INSTANCE.logIn( "Alice" );

doInJPA( this::entityManagerFactory, entityManager -> {

    Person person = new Person();
    person.setId( 1L );
    person.setFirstName( "John" );
    person.setLastName( "Doe" );

    entityManager.persist( person );
} );

CurrentUser.INSTANCE.logOut();
```

JAVA

```
INSERT INTO Person
(
    createdBy,
    firstName,
    lastName,
    updatedBy,
    id
)
VALUES
(?, ?, ?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Alice]
-- binding parameter [2] as [VARCHAR] - [John]
-- binding parameter [3] as [VARCHAR] - [Doe]
-- binding parameter [4] as [VARCHAR] - [Alice]
-- binding parameter [5] as [BIGINT]  - [1]
```

SQL

The same goes when the `Person` entity is updated. Hibernate is going to populate the `updatedBy` column with the currently logged user.

Example 60. `@Generated` update example

```
CurrentUser.INSTANCE.logIn( "Bob" );

doInJPA( this::entityManagerFactory, entityManager -> {
    Person person = entityManager.find( Person.class, 1L );
    person.setFirstName( "Mr. John" );
} );

CurrentUser.INSTANCE.logOut();
```

JAVA

```
SQL

UPDATE Person
SET
    createdBy = ?,
    firstName = ?,
    lastName = ?,
    updatedBy = ?
WHERE
    id = ?

-- binding parameter [1] as [VARCHAR] - [Alice]
-- binding parameter [2] as [VARCHAR] - [Mr. John]
-- binding parameter [3] as [VARCHAR] - [Doe]
-- binding parameter [4] as [VARCHAR] - [Bob]
-- binding parameter [5] as [BIGINT]  - [1]
```

`@CreationTimestamp` annotation

The `@CreationTimestamp` annotation instructs Hibernate to set the annotated entity attribute with the current timestamp value of the JVM when the entity is being persisted.

The supported property types are:

- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

Example 61. `@CreationTimestamp` mapping example

```java
@Entity(name = "Event")
public static class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`timestamp`")
    @CreationTimestamp
    private Date timestamp;

    public Event() {}

    public Long getId() {
        return id;
    }

    public Date getTimestamp() {
        return timestamp;
    }
}
```

When the `Event` entity is persisted, Hibernate is going to populate the underlying `timestamp` column with the current JVM timestamp value:

Example 62. `@CreationTimestamp` *persist example*

```java
Event dateEvent = new Event( );
entityManager.persist( dateEvent );
```

```sql
INSERT INTO Event("timestamp", id)
VALUES (?, ?)

-- binding parameter [1] as [TIMESTAMP] - [Tue Nov 15 16:24:20 EET 2016]
-- binding parameter [2] as [BIGINT]    - [1]
```

`@ValueGenerationType` meta-annotation

Hibernate 4.3 introduced the `@ValueGenerationType` meta-annotation, which is a new approach to declaring generated attributes or customizing generators.

`@Generated` has been retrofitted to use the `@ValueGenerationType` meta-annotation. But `@ValueGenerationType` exposes more features than what `@Generated` currently supports, and, to leverage some of those features, you'd simply wire up a new generator annotation.

As you'll see in the following examples, the `@ValueGenerationType` meta-annotation is used when declaring the custom annotation used to mark the entity properties that need a specific generation strategy. The actual generation logic must be implemented in class that implements the `AnnotationValueGeneration` interface.

Database-generated values

For example, let's say we want the timestamps to be generated by calls to the standard ANSI SQL function `current_timestamp` (rather than triggers or DEFAULT values):

Example 63. A `ValueGenerationType` mapping for database generation

JAVA

```java
@Entity(name = "Event")
public static class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`timestamp`")
    @FunctionCreationTimestamp
    private Date timestamp;

    public Event() {}

    public Long getId() {
        return id;
    }

    public Date getTimestamp() {
        return timestamp;
    }
}

@ValueGenerationType(generatedBy = FunctionCreationValueGeneration.class)
@Retention(RetentionPolicy.RUNTIME)
public @interface FunctionCreationTimestamp {}

public static class FunctionCreationValueGeneration
        implements AnnotationValueGeneration<FunctionCreationTimestamp> {

    @Override
    public void initialize(FunctionCreationTimestamp annotation, Class<?> propertyType) {
    }

    /**
     * Generate value on INSERT
     * @return when to generate the value
     */
    public GenerationTiming getGenerationTiming() {
        return GenerationTiming.INSERT;
    }

    /**
     * Returns null because the value is generated by the database.
     * @return null
     */
    public ValueGenerator<?> getValueGenerator() {
        return null;
    }

    /**
     * Returns true because the value is generated by the database.
     * @return true
     */
    public boolean referenceColumnInSql() {
        return true;
    }
```

```
    /**
     * Returns the database-generated value
     * @return database-generated value
     */
    public String getDatabaseGeneratedReferencedColumnValue() {
        return "current_timestamp";
    }
}
```

When persisting an `Event` entity, Hibernate generates the following SQL statement:

```
INSERT INTO Event ("timestamp", id)
VALUES (current_timestamp, 1)
```
SQL

As you can see, the `current_timestamp` value was used for assigning the `timestamp` column value.

In-memory-generated values

If the timestamp value needs to be generated in-memory, the following mapping must be used instead:

Example 64. A `ValueGenerationType` mapping for in-memory value generation

JAVA

```java
@Entity(name = "Event")
public static class Event {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`timestamp`")
    @FunctionCreationTimestamp
    private Date timestamp;

    public Event() {}

    public Long getId() {
        return id;
    }

    public Date getTimestamp() {
        return timestamp;
    }
}

@ValueGenerationType(generatedBy = FunctionCreationValueGeneration.class)
@Retention(RetentionPolicy.RUNTIME)
public @interface FunctionCreationTimestamp {}

public static class FunctionCreationValueGeneration
        implements AnnotationValueGeneration<FunctionCreationTimestamp> {

    @Override
    public void initialize(FunctionCreationTimestamp annotation, Class<?> propertyType) {
    }

    /**
     * Generate value on INSERT
     * @return when to generate the value
     */
    public GenerationTiming getGenerationTiming() {
        return GenerationTiming.INSERT;
    }

    /**
     * Returns the in-memory generated value
     * @return {@code true}
     */
    public ValueGenerator<?> getValueGenerator() {
        return (session, owner) -> new Date( );
    }

    /**
     * Returns false because the value is generated by the database.
     * @return false
     */
    public boolean referenceColumnInSql() {
        return false;
    }
```

```
    /**
     * Returns null because the value is generated in-memory.
     * @return null
     */
    public String getDatabaseGeneratedReferencedColumnValue() {
        return null;
    }
}
```

When persisting an `Event` entity, Hibernate generates the following SQL statement:

```
INSERT INTO Event ("timestamp", id)
VALUES ('Tue Mar 01 10:58:18 EET 2016', 1)
```
SQL

As you can see, the `new Date()` object value was used for assigning the `timestamp` column value.

2.3.19. Column transformers: read and write expressions

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to `@Basic` types. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like in the following example.

Example 65. `@ColumnTransformer` *example*

```java
@Entity(name = "Employee")
public static class Employee {

    @Id
    private Long id;

    @NaturalId
    private String username;

    @Column(name = "pswd")
    @ColumnTransformer(
        read = "decrypt( 'AES', '00', pswd )",
        write = "encrypt('AES', '00', ?)"
    )
    private String password;

    private int accessLevel;

    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;

    @ManyToMany(mappedBy = "employees")
    private List<Project> projects = new ArrayList<>();

    //Getters and setters omitted for brevity
}
```

> You can use the plural form `@ColumnTransformers` if more than one columns need to define either of these rules.

If a property uses more than one column, you must use the `forColumn` attribute to specify which column, the expressions are targeting.

Example 66. `@ColumnTransformer forColumn` *attribute usage*

```java
@Entity(name = "Savings")
public static class Savings {

    @Id
    private Long id;

    @Type(type = "org.hibernate.userguide.mapping.basic.MonetaryAmountUserType")
    @Columns(columns = {
        @Column(name = "money"),
        @Column(name = "currency")
    })
    @ColumnTransformer(
        forColumn = "money",
        read = "money / 100",
        write = "? * 100"
    )
    private MonetaryAmount wallet;

    //Getters and setters omitted for brevity

}
```

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

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

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

Example 67. Persisting an entity with a `@ColumnTransformer` and a composite type

```java
doInJPA( this::entityManagerFactory, entityManager -> {
    Savings savings = new Savings( );
    savings.setId( 1L );
    savings.setWallet( new MonetaryAmount( BigDecimal.TEN, Currency.getInstance( Locale.US ) ) );
    entityManager.persist( savings );
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    Savings savings = entityManager.find( Savings.class, 1L );
    assertEquals( 10, savings.getWallet().getAmount().intValue());
} );
```

```sql
INSERT INTO Savings (money, currency, id)
VALUES (10 * 100, 'USD', 1)

SELECT
    s.id as id1_0_0_,
    s.money / 100 as money2_0_0_,
    s.currency as currency3_0_0_
FROM
    Savings s
WHERE
    s.id = 1
```

2.3.20. `@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)

> You should be aware that the `@Formula` annotation takes a native SQL clause which can affect database portability.

Example 68. `@Formula` *mapping usage*

```java
@Entity(name = "Account")
public static class Account {

    @Id
    private Long id;

    private Double credit;

    private Double rate;

    @Formula(value = "credit * rate")
    private Double interest;

    //Getters and setters omitted for brevity

}
```

When loading the `Account` entity, Hibernate is going to calculate the `interest` property using the configured `@Formula`:

Example 69. Persisting an entity with a `@Formula` mapping

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    Account account = new Account( );
    account.setId( 1L );
    account.setCredit( 5000d );
    account.setRate( 1.25 / 100 );
    entityManager.persist( account );
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    Account account = entityManager.find( Account.class, 1L );
    assertEquals( Double.valueOf( 62.5d ), account.getInterest());
} );
```

SQL

```
INSERT INTO Account (credit, rate, id)
VALUES (5000.0, 0.0125, 1)

SELECT
    a.id as id1_0_0_,
    a.credit as credit2_0_0_,
    a.rate as rate3_0_0_,
    a.credit * a.rate as formula0_0_
FROM
    Account a
WHERE
    a.id = 1
```

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

2.3.21. `@Where`

Sometimes, you want to filter out entities or collections using a custom SQL criteria. This can be achieved using the `@Where` annotation, which can be applied to entities and collections.

Example 70. `@Where` *mapping usage*

JAVA

```
public enum AccountType {
    DEBIT,
    CREDIT
}

@Entity(name = "Client")
public static class Client {

    @Id
    private Long id;

    private String name;

    @Where( clause = "account_type = 'DEBIT'")
    @OneToMany(mappedBy = "client")
    private List<Account> debitAccounts = new ArrayList<>( );

    @Where( clause = "account_type = 'CREDIT'")
    @OneToMany(mappedBy = "client")
    private List<Account> creditAccounts = new ArrayList<>( );

    //Getters and setters omitted for brevity

}

@Entity(name = "Account")
@Where( clause = "active = true" )
public static class Account {

    @Id
    private Long id;

    @ManyToOne
    private Client client;

    @Column(name = "account_type")
    @Enumerated(EnumType.STRING)
    private AccountType type;

    private Double amount;

    private Double rate;

    private boolean active;

    //Getters and setters omitted for brevity

}
```

If the database contains the following entities:

Example 71. Persisting an fetching entities with a `@Where` mapping

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {

    Client client = new Client();
    client.setId( 1L );
    client.setName( "John Doe" );
    entityManager.persist( client );

    Account account1 = new Account( );
    account1.setId( 1L );
    account1.setType( AccountType.CREDIT );
    account1.setAmount( 5000d );
    account1.setRate( 1.25 / 100 );
    account1.setActive( true );
    account1.setClient( client );
    client.getCreditAccounts().add( account1 );
    entityManager.persist( account1 );

    Account account2 = new Account( );
    account2.setId( 2L );
    account2.setType( AccountType.DEBIT );
    account2.setAmount( 0d );
    account2.setRate( 1.05 / 100 );
    account2.setActive( false );
    account2.setClient( client );
    client.getDebitAccounts().add( account2 );
    entityManager.persist( account2 );

    Account account3 = new Account( );
    account3.setType( AccountType.DEBIT );
    account3.setId( 3L );
    account3.setAmount( 250d );
    account3.setRate( 1.05 / 100 );
    account3.setActive( true );
    account3.setClient( client );
    client.getDebitAccounts().add( account3 );
    entityManager.persist( account3 );
} );
```

SQL

```
INSERT INTO Client (name, id)
VALUES ('John Doe', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 5000.0, 1, 0.0125, 'CREDIT', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (false, 0.0, 1, 0.0105, 'DEBIT', 2)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 250.0, 1, 0.0105, 'DEBIT', 3)
```

When executing an `Account` entity query, Hibernate is going to filter out all records that are not active.

Example 72. Query entities mapped with `@Where`

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    List<Account> accounts = entityManager.createQuery(
        "select a from Account a", Account.class)
    .getResultList();
    assertEquals( 2, accounts.size());
} );
```

SQL

```
SELECT
    a.id as id1_0_,
    a.active as active2_0_,
    a.amount as amount3_0_,
    a.client_id as client_i6_0_,
    a.rate as rate4_0_,
    a.account_type as account_5_0_
FROM
    Account a
WHERE ( a.active = true )
```

When fetching the `debitAccounts` or the `creditAccounts` collections, Hibernate is going to apply the `@Where` clause filtering criteria to the associated child entities.

Example 73. Traversing collections mapped with `@Where`

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    Client client = entityManager.find( Client.class, 1L );
    assertEquals( 1, client.getCreditAccounts().size() );
    assertEquals( 1, client.getDebitAccounts().size() );
} );
```

```sql
SELECT
    c.client_id as client_i6_0_0_,
    c.id as id1_0_0_,
    c.id as id1_0_1_,
    c.active as active2_0_1_,
    c.amount as amount3_0_1_,
    c.client_id as client_i6_0_1_,
    c.rate as rate4_0_1_,
    c.account_type as account_5_0_1_
FROM
    Account c
WHERE ( c.active = true and c.account_type = 'CREDIT' ) AND c.client_id = 1

SELECT
    d.client_id as client_i6_0_0_,
    d.id as id1_0_0_,
    d.id as id1_0_1_,
    d.active as active2_0_1_,
    d.amount as amount3_0_1_,
    d.client_id as client_i6_0_1_,
    d.rate as rate4_0_1_,
    d.account_type as account_5_0_1_
FROM
    Account d
WHERE ( d.active = true and d.account_type = 'DEBIT' ) AND d.client_id = 1
```

2.3.22. `@Filter`

The `@Filter` annotation is another way to filter out entities or collections using a custom SQL criteria, for both entities and collections. Unlike the `@Where` annotation, `@Filter` allows you to parameterize the filter clause at runtime.

Example 74. `@Filter` *mapping usage*

JAVA

```
public enum AccountType {
    DEBIT,
    CREDIT
}

@Entity(name = "Client")
public static class Client {

    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "client")
    @Filter(name="activeAccount", condition="active = :active")
    private List<Account> accounts = new ArrayList<>( );

    //Getters and setters omitted for brevity

}

@Entity(name = "Account")
@FilterDef(name="activeAccount", parameters=@ParamDef( name="active", type="boolean" ) )
@Filter(name="activeAccount", condition="active = :active")
public static class Account {

    @Id
    private Long id;

    @ManyToOne
    private Client client;

    @Column(name = "account_type")
    @Enumerated(EnumType.STRING)
    private AccountType type;

    private Double amount;

    private Double rate;

    private boolean active;

    //Getters and setters omitted for brevity

}
```

If the database contains the following entities:

Example 75. Persisting an fetching entities with a `@Filter` mapping

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {

    Client client = new Client();
    client.setId( 1L );
    client.setName( "John Doe" );
    entityManager.persist( client );

    Account account1 = new Account( );
    account1.setId( 1L );
    account1.setType( AccountType.CREDIT );
    account1.setAmount( 5000d );
    account1.setRate( 1.25 / 100 );
    account1.setActive( true );
    account1.setClient( client );
    client.getAccounts().add( account1 );
    entityManager.persist( account1 );

    Account account2 = new Account( );
    account2.setId( 2L );
    account2.setType( AccountType.DEBIT );
    account2.setAmount( 0d );
    account2.setRate( 1.05 / 100 );
    account2.setActive( false );
    account2.setClient( client );
    client.getAccounts().add( account2 );
    entityManager.persist( account2 );

    Account account3 = new Account( );
    account3.setType( AccountType.DEBIT );
    account3.setId( 3L );
    account3.setAmount( 250d );
    account3.setRate( 1.05 / 100 );
    account3.setActive( true );
    account3.setClient( client );
    client.getAccounts().add( account3 );
    entityManager.persist( account3 );
} );
```

SQL

```
INSERT INTO Client (name, id)
VALUES ('John Doe', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 5000.0, 1, 0.0125, 'CREDIT', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (false, 0.0, 1, 0.0105, 'DEBIT', 2)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 250.0, 1, 0.0105, 'DEBIT', 3)
```

By default, without explicitly enabling the filter, Hibernate is going to fetch all `Account` entities. If the filter is enabled and the filter parameter value is provided, then Hibernate is going to apply the filtering criteria to the associated `Account` entities.

Example 76. Query entities mapped with `@Filter`

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    List<Account> accounts = entityManager.createQuery(
        "select a from Account a", Account.class)
    .getResultList();
    assertEquals( 3, accounts.size());
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    log.infof( "Activate filter [%s]", "activeAccount");

    entityManager
        .unwrap( Session.class )
        .enableFilter( "activeAccount" )
        .setParameter( "active", true);

    List<Account> accounts = entityManager.createQuery(
        "select a from Account a", Account.class)
    .getResultList();
    assertEquals( 2, accounts.size());
} );
```

SQL

```
SELECT
    a.id as id1_0_,
    a.active as active2_0_,
    a.amount as amount3_0_,
    a.client_id as client_i6_0_,
    a.rate as rate4_0_,
    a.account_type as account_5_0_
FROM
    Account a

-- Activate filter [activeAccount]

SELECT
    a.id as id1_0_,
    a.active as active2_0_,
    a.amount as amount3_0_,
    a.client_id as client_i6_0_,
    a.rate as rate4_0_,
    a.account_type as account_5_0_
FROM
    Account a
WHERE
    a.active = true
```

Filters apply to entity queries, but not to direct fetching. Therefore, in the following example, the filter is not taken into consideration when fetching an entity from the Persistence Context.

Fetching entities mapped with `@Filter`

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    log.infof( "Activate filter [%s]", "activeAccount");

    entityManager
        .unwrap( Session.class )
        .enableFilter( "activeAccount" )
        .setParameter( "active", true);

    Account account = entityManager.find( Account.class, 2L );
    assertFalse( account.isActive() );
} );
```

SQL

```
SELECT
    a.id as id1_0_0_,
    a.active as active2_0_0_,
    a.amount as amount3_0_0_,
    a.client_id as client_i6_0_0_,
    a.rate as rate4_0_0_,
    a.account_type as account_5_0_0_,
    c.id as id1_1_1_,
    c.name as name2_1_1_
FROM
    Account a
LEFT OUTER JOIN
    Client c
        ON a.client_id=c.id
WHERE
    a.id = 2
```

As you can see from the example above, contrary to an entity query, the filter does not prevent the entity from being loaded.

Just like with entity queries, collections can be filtered as well, but only if the filter is explicitly enabled on the currently running Hibernate `Session`. This way, when fetching the `accounts` collections, Hibernate is going to apply the `@Filter` clause filtering criteria to the associated collection entries.

Example 77. Traversing collections mapped with `@Filter`

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    Client client = entityManager.find( Client.class, 1L );
    assertEquals( 3, client.getAccounts().size() );
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    log.infof( "Activate filter [%s]", "activeAccount");

    entityManager
        .unwrap( Session.class )
        .enableFilter( "activeAccount" )
        .setParameter( "active", true);

    Client client = entityManager.find( Client.class, 1L );
    assertEquals( 2, client.getAccounts().size() );
} );
```

SQL

```
SELECT
    c.id as id1_1_0_,
    c.name as name2_1_0_
FROM
    Client c
WHERE
    c.id = 1

SELECT
    a.id as id1_0_,
    a.active as active2_0_,
    a.amount as amount3_0_,
    a.client_id as client_i6_0_,
    a.rate as rate4_0_,
    a.account_type as account_5_0_
FROM
    Account a
WHERE
    a.client_id = 1

-- Activate filter [activeAccount]

SELECT
    c.id as id1_1_0_,
    c.name as name2_1_0_
FROM
    Client c
WHERE
    c.id = 1

SELECT
    a.id as id1_0_,
    a.active as active2_0_,
    a.amount as amount3_0_,
    a.client_id as client_i6_0_,
    a.rate as rate4_0_,
    a.account_type as account_5_0_
FROM
    Account a
WHERE
    accounts0_.active = true
    and a.client_id = 1
```

> The main advantage of `@Filter` over the `@Where` clause is that the filtering criteria can be customized at runtime.

> It's not possible to combine the `@Filter` and `@Cache` collection annotations. This limitation is due to ensuring consistency and because the filtering information is not stored in the second-level cache.
>
> If caching was allowed for a currently filtered collection, then the second-level cache would store only a subset of the whole collection. Afterward, every other Session will get the filtered collection from the cache, even if the Session-level filters have not been explicitly activated.
>
> For this reason, the second-level collection cache is limited to storing whole collections, and not subsets.

2.3.23. `@FilterJoinTable`

When using the `@Filter` annotation with collections, the filtering is done against the child entries (entities or embeddables). However, if you have a link table between the parent entity and the child table, then you need to use the `@FilterJoinTable` to filter child entries according to some column contained in the join table.

The `@FilterJoinTable` annotation can be, therefore, applied to a unidirectional `@OneToMany` collection as illustrate din the following mapping:

Example 78. `@FilterJoinTable` *mapping usage*

JAVA

```
public enum AccountType {
    DEBIT,
    CREDIT
}

@Entity(name = "Client")
@FilterDef(name="firstAccounts", parameters=@ParamDef( name="maxOrderId", type="int" ) )
@Filter(name="firstAccounts", condition="order_id <= :maxOrderId")
public static class Client {

    @Id
    private Long id;

    private String name;

    @OneToMany
    @OrderColumn(name = "order_id")
    @FilterJoinTable(name="firstAccounts", condition="order_id <= :maxOrderId")
    private List<Account> accounts = new ArrayList<>( );

    //Getters and setters omitted for brevity

}

@Entity(name = "Account")
public static class Account {

    @Id
    private Long id;

    @Column(name = "account_type")
    @Enumerated(EnumType.STRING)
    private AccountType type;

    private Double amount;

    private Double rate;

    private boolean active;

    //Getters and setters omitted for brevity

}
```

If the database contains the following entities:

Example 79. Persisting an fetching entities with a `@FilterJoinTable` mapping

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {

    Client client = new Client();
    client.setId( 1L );
    client.setName( "John Doe" );
    entityManager.persist( client );

    Account account1 = new Account( );
    account1.setId( 1L );
    account1.setType( AccountType.CREDIT );
    account1.setAmount( 5000d );
    account1.setRate( 1.25 / 100 );
    account1.setActive( true );
    client.getAccounts().add( account1 );
    entityManager.persist( account1 );

    Account account2 = new Account( );
    account2.setId( 2L );
    account2.setType( AccountType.DEBIT );
    account2.setAmount( 0d );
    account2.setRate( 1.05 / 100 );
    account2.setActive( false );
    client.getAccounts().add( account2 );
    entityManager.persist( account2 );

    Account account3 = new Account( );
    account3.setType( AccountType.DEBIT );
    account3.setId( 3L );
    account3.setAmount( 250d );
    account3.setRate( 1.05 / 100 );
    account3.setActive( true );
    client.getAccounts().add( account3 );
    entityManager.persist( account3 );
} );
```

```sql
INSERT INTO Client (name, id)
VALUES ('John Doe', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 5000.0, 1, 0.0125, 'CREDIT', 1)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (false, 0.0, 1, 0.0105, 'DEBIT', 2)

INSERT INTO Account (active, amount, client_id, rate, account_type, id)
VALUES (true, 250.0, 1, 0.0105, 'DEBIT', 3)

INSERT INTO Client_Account (Client_id, order_id, accounts_id)
VALUES (1, 0, 1)

INSERT INTO Client_Account (Client_id, order_id, accounts_id)
VALUES (1, 0, 1)

INSERT INTO Client_Account (Client_id, order_id, accounts_id)
VALUES (1, 1, 2)

INSERT INTO Client_Account (Client_id, order_id, accounts_id)
VALUES (1, 2, 3)
```

The collections can be filtered if the associated filter is enabled on the currently running Hibernate `Session`. This way, when fetching the `accounts` collections, Hibernate is going to apply the `@FilterJoinTable` clause filtering criteria to the associated collection entries.

Example 80. Traversing collections mapped with `@FilterJoinTable`

```java
doInJPA( this::entityManagerFactory, entityManager -> {
    Client client = entityManager.find( Client.class, 1L );
    assertEquals( 3, client.getAccounts().size());
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    log.infof( "Activate filter [%s]", "firstAccounts");

    Client client = entityManager.find( Client.class, 1L );

    entityManager
        .unwrap( Session.class )
        .enableFilter( "firstAccounts" )
        .setParameter( "maxOrderId", 1);

    assertEquals( 2, client.getAccounts().size());
} );
```

```sql
SELECT
    ca.Client_id as Client_i1_2_0_,
    ca.accounts_id as accounts2_2_0_,
    ca.order_id as order_id3_0_,
    a.id as id1_0_1_,
    a.active as active2_0_1_,
    a.amount as amount3_0_1_,
    a.rate as rate4_0_1_,
    a.account_type as account_5_0_1_
FROM
    Client_Account ca
INNER JOIN
    Account a
ON  ca.accounts_id=a.id
WHERE
    ca.Client_id = ?

-- binding parameter [1] as [BIGINT] - [1]

-- Activate filter [firstAccounts]

SELECT
    ca.Client_id as Client_i1_2_0_,
    ca.accounts_id as accounts2_2_0_,
    ca.order_id as order_id3_0_,
    a.id as id1_0_1_,
    a.active as active2_0_1_,
    a.amount as amount3_0_1_,
    a.rate as rate4_0_1_,
    a.account_type as account_5_0_1_
FROM
    Client_Account ca
INNER JOIN
    Account a
ON  ca.accounts_id=a.id
WHERE
    ca.order_id <= ?
    AND ca.Client_id = ?

-- binding parameter [1] as [INTEGER] - [1]
-- binding parameter [2] as [BIGINT] - [1]
```

2.3.24. `@Any` mapping

There is one more type of property mapping. The `@Any` mapping defines a polymorphic association to classes from multiple tables. This type of mapping requires more than one column. The first column contains the type of the associated entity. The remaining columns contain the identifier.

> It is impossible to specify a foreign key constraint for this kind of association. This is not the usual way of mapping polymorphic associations and you should use this only in special cases (e.g. 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. The `metaType` attribute allows the application to specify a custom type that maps database column values to persistent classes that have identifier properties of the type specified by `idType`. You must specify the mapping from values of the `metaType` to class names.

For the next examples, consider the following `Property` class hierarchy:

Example 81. `Property` *class hierarchy*

JAVA

```
public interface Property<T> {

    String getName();

    T getValue();
}


@Entity
@Table(name="integer_property")
public class IntegerProperty implements Property<Integer> {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Column(name = "`value`")
    private Integer value;

    public Long getId() {
        return id;
    }

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

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

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

    public Integer getValue() {
        return value;
    }

    public void setValue(Integer value) {
        this.value = value;
    }
}


@Entity
@Table(name="string_property")
public class StringProperty implements Property<String> {

    @Id
    private Long id;

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

```
    private String name;

    @Column(name = "`value`")
    private String value;

    public Long getId() {
        return id;
    }

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

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

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

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

A `PropertyHolder` can reference any such property, and, because each `Property` belongs to a separate table, the `@Any` annotation is, therefore, required.

Example 82. `@Any` *mapping usage*

JAVA

```
@Entity
@Table( name = "property_holder" )
public class PropertyHolder {

    @Id
    private Long id;

    @Any(
        metaDef = "PropertyMetaDef",
        metaColumn = @Column( name = "property_type" )
    )
    @JoinColumn( name = "property_id" )
    private Property property;

        //Getters and setters are omitted for brevity

}
```

SQL

```
CREATE TABLE property_holder (
    id BIGINT NOT NULL,
    property_type VARCHAR(255),
    property_id BIGINT,
    PRIMARY KEY ( id )
)
```

As you can see, there are two columns used to reference a `Property` instance: `property_id` and `property_type`. The `property_id` is used to match the `id` column of either the `string_property` or `integer_property` tables, while the `property_type` is used to match the `string_property` or the `integer_property` table.

The table resolving mapping is defined by the `metaDef` attribute which references an `@AnyMetaDef` mapping. Although the `@AnyMetaDef` mapping could be set right next to the `@Any` annotation, it's good practice to reuse it, therefore it makes sense to configure it on a class or package-level basis.

The `package-info.java` contains the `@AnyMetaDef` mapping:

Example 83. `@Any` mapping usage

```
@AnyMetaDef( name= "PropertyMetaDef", metaType = "string", idType = "long",
    metaValues = {
            @MetaValue(value = "S", targetEntity = StringProperty.class),
            @MetaValue(value = "I", targetEntity = IntegerProperty.class)
        }
    )
package org.hibernate.userguide.mapping.basic.any;

import org.hibernate.annotations.AnyMetaDef;
import org.hibernate.annotations.MetaValue;
```

JAVA

> It is recommended to place the `@AnyMetaDef` mapping as a package metadata.

To see how the `@Any` annotation in action, consider the following example:

Example 84. `@Any` *mapping usage*

JAVA

```
doInHibernate( this::sessionFactory, session -> {
    IntegerProperty ageProperty = new IntegerProperty();
    ageProperty.setId( 1L );
    ageProperty.setName( "age" );
    ageProperty.setValue( 23 );

    StringProperty nameProperty = new StringProperty();
    nameProperty.setId( 1L );
    nameProperty.setName( "name" );
    nameProperty.setValue( "John Doe" );

    session.persist( ageProperty );
    session.persist( nameProperty );

    PropertyHolder namePropertyHolder = new PropertyHolder();
    namePropertyHolder.setId( 1L );
    namePropertyHolder.setProperty( nameProperty );
    session.persist( namePropertyHolder );
} );

doInHibernate( this::sessionFactory, session -> {
    PropertyHolder propertyHolder = session.get( PropertyHolder.class, 1L );
    assertEquals("name", propertyHolder.getProperty().getName());
    assertEquals("John Doe", propertyHolder.getProperty().getValue());
} );
```

SQL

```
INSERT INTO integer_property
       ( "name", "value", id )
VALUES ( 'age', 23, 1 )

INSERT INTO string_property
       ( "name", "value", id )
VALUES ( 'name', 'John Doe', 1 )

INSERT INTO property_holder
       ( property_type, property_id, id )
VALUES ( 'S', 1, 1 )


SELECT ph.id AS id1_1_0_,
       ph.property_type AS property2_1_0_,
       ph.property_id AS property3_1_0_
FROM   property_holder ph
WHERE  ph.id = 1


SELECT sp.id AS id1_2_0_,
       sp."name" AS name2_2_0_,
       sp."value" AS value3_2_0_
FROM   string_property sp
WHERE  sp.id = 1
```

`@ManyToAny` mapping

The `@Any` mapping is useful to emulate a `@ManyToOne` association when there can be multiple target entities. To emulate a `@OneToMany` association, the `@ManyToAny` annotation must be used.

In the following example, the `PropertyRepository` entity has a collection of `Property` entities. The `repository_properties` link table holds the associations between `PropertyRepository` and `Property` entities.

Example 85. `@ManyToAny` mapping usage

JAVA

```
@Entity
@Table( name = "property_repository" )
public class PropertyRepository {

    @Id
    private Long id;

    @ManyToAny(
        metaDef = "PropertyMetaDef",
        metaColumn = @Column( name = "property_type" )
    )
    @Cascade( { org.hibernate.annotations.CascadeType.ALL })
    @JoinTable(name = "repository_properties",
        joinColumns = @JoinColumn(name = "repository_id"),
        inverseJoinColumns = @JoinColumn(name = "property_id")
    )
    private List<Property<?>> properties = new ArrayList<>(  );

        //Getters and setters are omitted for brevity

}
```

SQL

```
CREATE TABLE property_repository (
    id BIGINT NOT NULL,
    PRIMARY KEY ( id )
)

CREATE TABLE repository_properties (
    repository_id BIGINT NOT NULL,
    property_type VARCHAR(255),
    property_id BIGINT NOT NULL
)
```

To see how the `@ManyToAny` annotation works, consider the following example:

Example 86. `@Any` mapping usage

JAVA

```
doInHibernate( this::sessionFactory, session -> {
    IntegerProperty ageProperty = new IntegerProperty();
    ageProperty.setId( 1L );
    ageProperty.setName( "age" );
    ageProperty.setValue( 23 );

    StringProperty nameProperty = new StringProperty();
    nameProperty.setId( 1L );
    nameProperty.setName( "name" );
    nameProperty.setValue( "John Doe" );

    session.persist( ageProperty );
    session.persist( nameProperty );

    PropertyRepository propertyRepository = new PropertyRepository();
    propertyRepository.setId( 1L );
    propertyRepository.getProperties().add( ageProperty );
    propertyRepository.getProperties().add( nameProperty );
    session.persist( propertyRepository );
} );

doInHibernate( this::sessionFactory, session -> {
    PropertyRepository propertyRepository = session.get( PropertyRepository.class, 1L );
    assertEquals(2, propertyRepository.getProperties().size());
    for(Property property : propertyRepository.getProperties()) {
        assertNotNull( property.getValue() );
    }
} );
```

```SQL
INSERT INTO integer_property
       ( "name", "value", id )
VALUES ( 'age', 23, 1 )

INSERT INTO string_property
       ( "name", "value", id )
VALUES ( 'name', 'John Doe', 1 )

INSERT INTO property_repository ( id )
VALUES ( 1 )

INSERT INTO repository_properties
    ( repository_id , property_type , property_id )
VALUES
    ( 1 , 'I' , 1 )

INSERT INTO repository_properties
    ( repository_id , property_type , property_id )
VALUES
    ( 1 , 'S' , 1 )

SELECT pr.id AS id1_1_0_
FROM   property_repository pr
WHERE  pr.id = 1

SELECT ip.id AS id1_0_0_ ,
       integerpro0_."name" AS name2_0_0_ ,
       integerpro0_."value" AS value3_0_0_
FROM   integer_property integerpro0_
WHERE  integerpro0_.id = 1

SELECT sp.id AS id1_3_0_ ,
       sp."name" AS name2_3_0_ ,
       sp."value" AS value3_3_0_
FROM   string_property sp
WHERE  sp.id = 1
```

2.3.25. `@JoinFormula` mapping

The `@JoinFormula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/JoinFormula.html) annotation is used to customize the join between a child Foreign Key and a parent row Primary Key.

Example 87. `@JoinFormula` mapping usage

JAVA

```
@Entity(name = "User")
@Table(name = "users")
public static class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String phoneNumber;

    @ManyToOne
    @JoinFormula( "REGEXP_REPLACE(phoneNumber, '\\+(\\d+)-.*', '\\1')::int" )
    private Country country;

    //Getters and setters omitted for brevity

}

@Entity(name = "Country")
@Table(name = "countries")
public static class Country {

    @Id
    private Integer id;

    private String name;

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( !( o instanceof Country ) ) {
            return false;
        }
        Country country = (Country) o;
        return Objects.equals( getId(), country.getId() );
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash( getId() );
    }
}
```

SQL

```
CREATE TABLE countries (
    id int4 NOT NULL,
    name VARCHAR(255),
    PRIMARY KEY ( id )
)

CREATE TABLE users (
    id int8 NOT NULL,
    firstName VARCHAR(255),
    lastName VARCHAR(255),
    phoneNumber VARCHAR(255),
    PRIMARY KEY ( id )
)
```

The `country` association in the `User` entity is mapped by the country identifier provided by the `phoneNumber` property.

Considering we have the following entities:

Example 88. `@JoinFormula` *mapping usage*

```java
Country US = new Country();
US.setId( 1 );
US.setName( "United States" );

Country Romania = new Country();
Romania.setId( 40 );
Romania.setName( "Romania" );

doInJPA( this::entityManagerFactory, entityManager -> {
    entityManager.persist( US );
    entityManager.persist( Romania );
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    User user1 = new User( );
    user1.setId( 1L );
    user1.setFirstName( "John" );
    user1.setLastName( "Doe" );
    user1.setPhoneNumber( "+1-234-5678" );
    entityManager.persist( user1 );

    User user2 = new User( );
    user2.setId( 2L );
    user2.setFirstName( "Vlad" );
    user2.setLastName( "Mihalcea" );
    user2.setPhoneNumber( "+40-123-4567" );
    entityManager.persist( user2 );
} );
```

When fetching the `User` entities, the `country` property is mapped by the `@JoinFormula` expression:

Example 89. `@JoinFormula` *mapping usage*

```java
doInJPA( this::entityManagerFactory, entityManager -> {
    log.info( "Fetch User entities" );

    User john = entityManager.find( User.class, 1L );
    assertEquals( US, john.getCountry());

    User vlad = entityManager.find( User.class, 2L );
    assertEquals( Romania, vlad.getCountry());
} );
```

```
SQL
-- Fetch User entities

SELECT
    u.id as id1_1_0_,
    u.firstName as firstNam2_1_0_,
    u.lastName as lastName3_1_0_,
    u.phoneNumber as phoneNum4_1_0_,
    REGEXP_REPLACE(u.phoneNumber, '\+(\d+)-.*', '\1')::int as formula1_0_,
    c.id as id1_0_1_,
    c.name as name2_0_1_
FROM
    users u
LEFT OUTER JOIN
    countries c
        ON REGEXP_REPLACE(u.phoneNumber, '\+(\d+)-.*', '\1')::int = c.id
WHERE
    u.id=?

-- binding parameter [1] as [BIGINT] - [1]

SELECT
    u.id as id1_1_0_,
    u.firstName as firstNam2_1_0_,
    u.lastName as lastName3_1_0_,
    u.phoneNumber as phoneNum4_1_0_,
    REGEXP_REPLACE(u.phoneNumber, '\+(\d+)-.*', '\1')::int as formula1_0_,
    c.id as id1_0_1_,
    c.name as name2_0_1_
FROM
    users u
LEFT OUTER JOIN
    countries c
        ON REGEXP_REPLACE(u.phoneNumber, '\+(\d+)-.*', '\1')::int = c.id
WHERE
    u.id=?

-- binding parameter [1] as [BIGINT] - [2]
```

Therefore, the `@JoinFormula` annotation is used to define a custom join association between the parent-child association.

2.3.26. `@JoinColumnOrFormula` mapping

The `@JoinColumnOrFormula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/JoinColumnOrFormula.html) annotation is used to customize the join between a child Foreign Key and a parent row Primary Key when we need to tak into consideration a column value as well as a `@JoinFormula` .

Example 90. `@JoinColumnOrFormula` *mapping usage*

JAVA

```
@Entity(name = "User")
@Table(name = "users")
public static class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String language;

    @ManyToOne
    @JoinColumnOrFormula( column =
        @JoinColumn(
            name = "language",
            referencedColumnName = "primaryLanguage",
            insertable = false,
            updatable = false
        )
    )
    @JoinColumnOrFormula( formula =
        @JoinFormula(
            value = "true",
            referencedColumnName = "is_default"
        )
    )
    private Country country;

    //Getters and setters omitted for brevity

}

@Entity(name = "Country")
@Table(name = "countries")
public static class Country implements Serializable {

    @Id
    private Integer id;

    private String name;

    private String primaryLanguage;

    @Column(name = "is_default")
    private boolean _default;

    public int getId() {
        return id;
    }

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

```
    public String getName() {
        return name;
    }

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

    public String getPrimaryLanguage() {
        return primaryLanguage;
    }

    public void setPrimaryLanguage(String primaryLanguage) {
        this.primaryLanguage = primaryLanguage;
    }

    public boolean isDefault() {
        return _default;
    }

    public void setDefault(boolean _default) {
        this._default = _default;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( !( o instanceof Country ) ) {
            return false;
        }
        Country country = (Country) o;
        return Objects.equals( getId(), country.getId() );
    }

    @Override
    public int hashCode() {
        return Objects.hash( getId() );
    }
}
```

```SQL
CREATE TABLE countries (
    id INTEGER NOT NULL,
    is_default boolean,
    name VARCHAR(255),
    primaryLanguage VARCHAR(255),
    PRIMARY KEY ( id )
)

CREATE TABLE users (
    id BIGINT NOT NULL,
    firstName VARCHAR(255),
    language VARCHAR(255),
    lastName VARCHAR(255),
    PRIMARY KEY ( id )
)
```

The `country` association in the `User` entity is mapped by the `language` property value and the the associated `Country` `is_default` column value.

Considering we have the following entities:

Example 91. `@JoinColumnOrFormula` *mapping usage*

```java
Country US = new Country();
US.setId( 1 );
US.setDefault( true );
US.setPrimaryLanguage( "English" );
US.setName( "United States" );

Country Romania = new Country();
Romania.setId( 40 );
Romania.setDefault( true );
Romania.setName( "Romania" );
Romania.setPrimaryLanguage( "Romanian" );

doInJPA( this::entityManagerFactory, entityManager -> {
    entityManager.persist( US );
    entityManager.persist( Romania );
} );

doInJPA( this::entityManagerFactory, entityManager -> {
    User user1 = new User( );
    user1.setId( 1L );
    user1.setFirstName( "John" );
    user1.setLastName( "Doe" );
    user1.setLanguage( "English" );
    entityManager.persist( user1 );

    User user2 = new User( );
    user2.setId( 2L );
    user2.setFirstName( "Vlad" );
    user2.setLastName( "Mihalcea" );
    user2.setLanguage( "Romanian" );
    entityManager.persist( user2 );

} );
```

When fetching the `User` entities, the `country` property is mapped by the `@JoinColumnOrFormula` expression:

Example 92. `@JoinColumnOrFormula` *mapping usage*

```java
doInJPA( this::entityManagerFactory, entityManager -> {
    log.info( "Fetch User entities" );

    User john = entityManager.find( User.class, 1L );
    assertEquals( US, john.getCountry());

    User vlad = entityManager.find( User.class, 2L );
    assertEquals( Romania, vlad.getCountry());
} );
```

```sql
SELECT
    u.id as id1_1_0_,
    u.language as language3_1_0_,
    u.firstName as firstNam2_1_0_,
    u.lastName as lastName4_1_0_,
    1 as formula1_0_,
    c.id as id1_0_1_,
    c.is_default as is_defau2_0_1_,
    c.name as name3_0_1_,
    c.primaryLanguage as primaryL4_0_1_
FROM
    users u
LEFT OUTER JOIN
    countries c
        ON u.language = c.primaryLanguage
        AND 1 = c.is_default
WHERE
    u.id = ?

-- binding parameter [1] as [BIGINT] - [1]

SELECT
    u.id as id1_1_0_,
    u.language as language3_1_0_,
    u.firstName as firstNam2_1_0_,
    u.lastName as lastName4_1_0_,
    1 as formula1_0_,
    c.id as id1_0_1_,
    c.is_default as is_defau2_0_1_,
    c.name as name3_0_1_,
    c.primaryLanguage as primaryL4_0_1_
FROM
    users u
LEFT OUTER JOIN
    countries c
        ON u.language = c.primaryLanguage
        AND 1 = c.is_default
WHERE
    u.id = ?

-- binding parameter [1] as [BIGINT] - [2]
```

Therefore, the `@JoinColumnOrFormula` annotation is used to define a custom join association between the parent-child association.

2.4. Embeddable types

Historically Hibernate called these components. JPA calls them embeddables. Either way the concept is the same: a composition of values. For example we might have a Name class that is a composition of first-name and last-name, or an Address class that is a composition of street, city, postal code, etc.

> *Usage of the word embeddable*
>
> To avoid any confusion with the annotation that marks a given embeddable type, the annotation will be further referred as `@Embeddable`.
>
> Throughout this chapter and thereafter, for brevity sake, embeddable types may also be referred as *embeddable*.

Example 93. Simple embeddable type example

JAVA

```
@Embeddable
public class Name {

    private String firstName;

    private String middleName;

    private String lastName;

    ...
}
```

JAVA

```
@Embeddable
public class Address {

    private String line1;

    private String line2;

    @Embedded
    private ZipCode zipCode;

    ...

    @Embeddable
    public static class Zip {

        private String postalCode;

        private String plus4;

        ...
    }
}
```

An embeddable type is another form of value type, and its lifecycle is bound to a parent entity type, therefore inheriting the attribute access from its parent (for details on attribute access, see Access strategies).

Embeddable types can be made up of basic values as well as associations, with the caveat that, when used as collection elements, they cannot define collections themselves.

2.4.1. Component / Embedded

Most often, embeddable types are used to group multiple basic type mappings and reuse them across several entities.

Example 94. Simple Embeddedable

```java
@Entity
public class Person {

    @Id
    private Integer id;

    @Embedded
    private Name name;

    ...
}
```

> JPA defines two terms for working with an embeddable type: `@Embeddable` and `@Embedded`. `@Embeddable` is used to describe the mapping type itself (e.g. `Name`). `@Embedded` is for referencing a given embeddable type (e.g. `person.name`).

So, the embeddable type is represented by the `Name` class and the parent makes use of it through the `person.name` object composition.

Example 95. Person table

```
SQL
create table Person (
    id integer not null,
    firstName VARCHAR,
    middleName VARCHAR,
    lastName VARCHAR,
    ...
)
```

The composed values are mapped to the same table as the parent table. Composition is part of good OO data modeling (idiomatic Java). In fact, that table could also be mapped by the following entity type instead.

Example 96. Alternative to embeddable type composition

```
JAVA
@Entity
public class Person {

    @Id
    private Integer id;

    private String firstName;

    private String middleName;

    private String lastName;

    ...
}
```

The composition form is certainly more Object-oriented, and that becomes more evident as we work with multiple embeddable types.

2.4.2. Multiple embeddable types

Example 97. Multiple embeddable types

```java
@Entity
public class Contact {

    @Id
    private Integer id;

    @Embedded
    private Name name;

    @Embedded
    private Address homeAddress;

    @Embedded
    private Address mailingAddress;

    @Embedded
    private Address workAddress;

    ...
}
```

Although from an object-oriented perspective, it's much more convenient to work with embeddable types, this example doesn't work as-is. When the same embeddable type is included multiple times in the same parent entity type, the JPA specification demands setting the associated column names explicitly.

This requirement is due to how object properties are mapped to database columns. By default, JPA expects a database column having the same name with its associated object property. When including multiple embeddables, the implicit name-based mapping rule doesn't work anymore because multiple object properties could end-up being mapped to the same database column.

We have a few options to handle this issue.

2.4.3. JPA's AttributeOverride

JPA defines the `@AttributeOverride` annotation to handle this scenario.

Example 98. JPA's AttributeOverride

JAVA

```
@Entity
public class Contact {

    @Id
    private Integer id;

    @Embedded
    private Name name;

    @Embedded
    @AttributeOverrides(
        @AttributeOverride(
            name = "line1",
            column = @Column( name = "home_address_line1" ),
        ),
        @AttributeOverride(
            name = "line2",
            column = @Column( name = "home_address_line2" )
        ),
        @AttributeOverride(
            name = "zipCode.postalCode",
            column = @Column( name = "home_address_postal_cd" )
        ),
        @AttributeOverride(
            name = "zipCode.plus4",
            column = @Column( name = "home_address_postal_plus4" )
        )
    )
    private Address homeAddress;

    @Embedded
    @AttributeOverrides(
        @AttributeOverride(
            name = "line1",
            column = @Column( name = "mailing_address_line1" ),
        ),
        @AttributeOverride(
            name = "line2",
            column = @Column( name = "mailing_address_line2" )
        ),
        @AttributeOverride(
            name = "zipCode.postalCode",
            column = @Column( name = "mailing_address_postal_cd" )
        ),
        @AttributeOverride(
            name = "zipCode.plus4",
            column = @Column( name = "mailing_address_postal_plus4" )
        )
    )
    private Address mailingAddress;

    @Embedded
    @AttributeOverrides(
        @AttributeOverride(
            name = "line1",
            column = @Column( name = "work_address_line1" ),
```

```
        ),
        @AttributeOverride(
            name = "line2",
            column = @Column( name = "work_address_line2" )
        ),
        @AttributeOverride(
            name = "zipCode.postalCode",
            column = @Column( name = "work_address_postal_cd" )
        ),
        @AttributeOverride(
            name = "zipCode.plus4",
            column = @Column( name = "work_address_postal_plus4" )
        )
    )
    private Address workAddress;

    ...
}
```

This way, the mapping conflict is resolved by setting up explicit name-based property-column type mappings.

2.4.4. ImplicitNamingStrategy

> This is a Hibernate specific feature. Users concerned with JPA provider portability should instead prefer explicit column naming with `@AttributeOverride`.

Hibernate naming strategies are covered in detail in Naming. However, for the purposes of this discussion, Hibernate has the capability to interpret implicit column names in a way that is safe for use with multiple embeddable types.

Example 99. Enabling embeddable type safe implicit naming

```
MetadataSources sources = ...;
sources.addAnnotatedClass( Address.class );
sources.addAnnotatedClass( Name.class );
sources.addAnnotatedClass( Contact.class );

Metadata metadata = sources.getMetadataBuilder().applyImplicitNamingStrategy(
ImplicitNamingStrategyComponentPathImpl.INSTANCE )
...
.build();
```

JAVA

```
create table Contact(
    id integer not null,
    name_firstName VARCHAR,
    name_middleName VARCHAR,
    name_lastName VARCHAR,
    homeAddress_line1 VARCHAR,
    homeAddress_line2 VARCHAR,
    homeAddress_zipCode_postalCode VARCHAR,
    homeAddress_zipCode_plus4 VARCHAR,
    mailingAddress_line1 VARCHAR,
    mailingAddress_line2 VARCHAR,
    mailingAddress_zipCode_postalCode VARCHAR,
    mailingAddress_zipCode_plus4 VARCHAR,
    workAddress_line1 VARCHAR,
    workAddress_line2 VARCHAR,
    workAddress_zipCode_postalCode VARCHAR,
    workAddress_zipCode_plus4 VARCHAR,
    ...
)
```

SQL

Now the "path" to attributes are used in the implicit column naming. You could even develop your own to do special implicit naming.

2.4.5. Collections of embeddable types

Collections of embeddable types are specifically value collections (as embeddable types are a value type). Value collections are covered in detail in Collections of value types.

2.4.6. Embeddable types as Map key

Embeddable types can also be used as `Map` keys. This topic is converted in detail in Map - key.

2.4.7. Embeddable types as identifiers

Embeddable types can also be used as entity type identifiers. This usage is covered in detail in Composite identifiers.

> Embeddable types that are used as collection entries, map keys or entity type identifiers cannot include their own collection mappings.

2.5. Entity types

> *Usage of the word entity*
>
> The entity type describes the mapping between the actual persistable domain model object and a database table row. To avoid any confusion with the annotation that marks a given entity type, the annotation will be further referred as `@Entity`.
>
> Throughout this chapter and thereafter, entity types will be simply referred as *entity*.

2.5.1. POJO Models

Section *2.1 The Entity Class* of the *JPA 2.1 specification* defines its requirements for an entity class. Applications that wish to remain portable across JPA providers should adhere to these requirements.

- The entity class must be annotated with the `javax.persistence.Entity` annotation (or be denoted as such in XML mapping)
- The entity class must have a public or protected no-argument constructor. It may define additional constructors as well.
- The entity class must be a top-level class.
- An enum or interface may not be designated as an entity.
- The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
- If an entity instance is to be used remotely as a detached object, the entity class must implement the `Serializable` interface.
- Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.
- The persistent state of an entity is represented by instance variables, which may correspond to JavaBean-style properties. An instance variable must be directly accessed only from within the methods of the entity by the entity instance itself. The state of the entity is available to clients only through the entity's accessor methods (getter/setter methods) or other business methods.

Hibernate, however, is not as strict in its requirements. The differences from the list above include:

- The entity class must have a no-argument constructor, which may be public, protected or package visibility. It may define additional constructors as well.
- The entity class *need not* be a top-level class.
- Technically Hibernate can persist final classes or classes with final persistent state accessor (getter/setter) methods. However, it is generally not a good idea as doing so will stop Hibernate from being able to generate proxies for lazy-loading the entity.
- Hibernate does not restrict the application developer from exposing instance variables and reference them from outside the entity class itself. The validity of such a paradigm, however, is debatable at best.

Let's look at each requirement in detail.

2.5.2. Prefer non-final classes

A central feature of Hibernate is the ability to load lazily certain entity instance variables (attributes) via runtime proxies. This feature depends upon the entity class being non-final or else implementing an interface that declares all the attribute getters/setters. You can still persist final classes that do not implement such an interface with Hibernate, but you will not be able to use proxies for fetching lazy associations, therefore limiting your options for performance tuning. For the very same reason, you should also avoid declaring persistent attribute getters and setters as final.

> Starting in 5.0 Hibernate offers a more robust version of bytecode enhancement as another means for handling lazy loading. Hibernate had some bytecode re-writing capabilities prior to 5.0 but they were very rudimentary. See the BytecodeEnhancement for additional information on fetching and on bytecode enhancement.

2.5.3. Implement a no-argument constructor

The entity class should have a no-argument constructor. Both Hibernate and JPA require this.

JPA requires that this constructor be defined as public or protected. Hibernate, for the most part, does not care about the constructor visibility, as long as the system SecurityManager allows overriding the visibility setting. That said, the constructor should be defined with at least package visibility if you wish to leverage runtime proxy generation.

2.5.4. Declare getters and setters for persistent attributes

The JPA specification requires this, otherwise the model would prevent accessing the entity persistent state fields directly from outside the entity itself.

Although Hibernate does not require it, it is recommended to follow the JavaBean conventions and define getters and setters for entity persistent attributes. Nevertheless, you can still tell Hibernate to directly access the entity fields.

Attributes (whether fields or getters/setters) need not be declared public. Hibernate can deal with attributes declared with public, protected, package or private visibility. Again, if wanting to use runtime proxy generation for lazy loading, the getter/setter should grant access to at least package visibility.

2.5.5. Provide identifier attribute(s)

> Historically this was considered optional. However, not defining identifier attribute(s) on the entity should be considered a deprecated feature that will be removed in an upcoming release.

The identifier attribute does not necessarily need to be mapped to the column(s) that physically define the primary key. However, it should map to column(s) that can uniquely identify each row.

> We recommend that you declare consistently-named identifier attributes on persistent classes and that you use a nullable (i.e., non-primitive) type.

The placement of the `@Id` annotation marks the persistence state access strategy.

Example 100. Identifier

```
@Id
private Integer id;
```
JAVA

Hibernate offers multiple identifier generation strategies, see the Identifier Generators chapter for more about this topic.

2.5.6. Mapping the entity

The main piece in mapping the entity is the `javax.persistence.Entity` annotation. The `@Entity` annotation defines just one attribute `name` which is used to give a specific entity name for use in JPQL queries. By default, the entity name represents the unqualified name of the entity class itself.

Example 101. Simple `@Entity`

```
JAVA
@Entity
public class Simple {
    ...
}
```

An entity models a database table. The identifier uniquely identifies each row in that table. By default, the name of the table is assumed to be the same as the name of the entity. To explicitly give the name of the table or to specify other information about the table, we would use the `javax.persistence.Table` annotation.

Example 102. Simple `@Entity` *with* `@Table`

```
JAVA
@Entity
@Table( catalog = "CRM", schema = "purchasing", name = "t_simple" )
public class Simple {
    ...
}
```

2.5.7. Implementing `equals()` and `hashCode()`

> Much of the discussion in this section deals with the relation of an entity to a Hibernate Session, whether the entity is managed, transient or detached. If you are unfamiliar with these topics, they are explained in the Persistence Context chapter.

Whether to implement `equals()` and `hashCode()` methods in your domain model, let alone how to implement them, is a surprisingly tricky discussion when it comes to ORM.

There is really just one absolute case: a class that acts as an identifier must implement equals/hashCode based on the id value(s). Generally, this is pertinent for user-defined classes used as composite identifiers. Beyond this one very specific use case and few others we will discuss below, you may want to consider not implementing equals/hashCode altogether.

So what's all the fuss? Normally, most Java objects provide a built-in `equals()` and `hashCode()` based on the object's identity, so each new object will be different from all others. This is generally what you want in ordinary Java programming. Conceptually however this starts to break down when you start to think about the possibility of multiple instances of a class representing the same data.

This is, in fact, exactly the case when dealing with data coming from a database. Every time we load a specific `Person` from the database we would naturally get a unique instance. Hibernate, however, works hard to make sure that does not happen within a given `Session`. In fact, Hibernate guarantees equivalence of persistent identity (database row) and Java identity inside a particular session scope. So if we ask a Hibernate `Session` to load that specific Person multiple times we will actually get back the same *instance*:

Example 103. Scope of identity

JAVA

```
Session session=...;

Person p1 = session.get( Person.class,1 );
Person p2 = session.get( Person.class,1 );

// this evaluates to true
assert p1==p2;
```

Consider another example using a persistent `java.util.Set`:

Example 104. Set usage with Session-scoped identity

JAVA

```
Session session=...;

Club club = session.get( Club.class,1 );

Person p1 = session.get( Person.class,1 );
Person p2 = session.get( Person.class,1 );

club.getMembers().add( p1 );
club.getMembers().add( p2 );

// this evaluates to true
assert club.getMembers.size()==1;
```

However, the semantic changes when we mix instances loaded from different Sessions:

Example 105. Mixed Sessions

```
JAVA
Session session1=...;
Session session2=...;

Person p1 = session1.get( Person.class,1 );
Person p2 = session2.get( Person.class,1 );

// this evaluates to false
assert p1==p2;
```

```
JAVA
Session session1=...;
Session session2=...;

Club club = session1.get( Club.class,1 );

Person p1 = session1.get( Person.class,1 );
Person p2 = session2.get( Person.class,1 );

club.getMembers().add( p1 );
club.getMembers().add( p2 );

// this evaluates to ... well it depends
assert club.getMembers.size()==1;
```

Specifically the outcome in this last example will depend on whether the `Person` class implemented equals/hashCode, and, if so, how.

Consider yet another case:

Example 106. Sets with transient entities

```
JAVA
Session session=...;

Club club = session.get( Club.class,1 );

Person p1 = new Person(...);
Person p2 = new Person(...);

club.getMembers().add( p1 );
club.getMembers().add( p2 );

// this evaluates to ... again, it depends
assert club.getMembers.size()==1;
```

In cases where you will be dealing with entities outside of a Session (whether they be transient or detached), especially in cases where you will be using them in Java collections, you should consider implementing equals/hashCode.

A common initial approach is to use the entity's identifier attribute as the basis for equals/hashCode calculations:

Example 107. Naive equals/hashCode implementation

JAVA

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Integer id;

    @Override
    public int hashCode() {
        return Objects.hash( id );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( !( o instanceof Person ) ) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals( id, person.id );
    }
}
```

It turns out that this still breaks when adding transient instance of `Person` to a set as we saw in the last example:

Example 108. Still trouble

```java
Session session=...;
session.getTransaction().begin();

Club club = session.get( Club.class,1 );

Person p1 = new Person(...);
Person p2 = new Person(...);

club.getMembers().add( p1 );
club.getMembers().add( p2 );

session.getTransaction().commit();

// will actually resolve to false!
assert club.getMembers().contains( p1 );
```

The issue here is a conflict between the use of generated identifier, the contract of `Set` and the equals/hashCode implementations. `Set` says that the equals/hashCode value for an object should not change while the object is part of the `Set`. But that is exactly what happened here because the equals/hasCode are based on the (generated) id, which was not set until the `session.getTransaction().commit()` call.

Note that this is just a concern when using generated identifiers. If you are using assigned identifiers this will not be a problem, assuming the identifier value is assigned prior to adding to the `Set`.

Another option is to force the identifier to be generated and set prior to adding to the `Set`:

Example 109. Forcing identifier generation

```java
Session session=...;
session.getTransaction().begin();

Club club = session.get( Club.class,1 );

Person p1 = new Person(...);
Person p2 = new Person(...);

session.save( p1 );
session.save( p2 );
session.flush();

club.getMembers().add( p1 );
club.getMembers().add( p2 );

session.getTransaction().commit();

// will actually resolve to false!
assert club.getMembers().contains( p1 );
```

But this is often not feasible.

The final approach is to use a "better" equals/hashCode implementation, making use of a natural-id or business-key.

Example 110. Better equals/hashCode with natural-id

JAVA

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Integer id;

    @NaturalId
    private String ssn;

    protected Person() {
        // Constructor for ORM
    }

    public Person( String ssn ) {
        // Constructor for app
        this.ssn = ssn;
    }

    @Override
    public int hashCode() {
        return Objects.hash( ssn );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( !( o instanceof Person ) ) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals( ssn, person.ssn );
    }
}
```

As you can see the question of equals/hashCode is not trivial, nor is there a one-size-fits-all solution.

> Although using a natural-id is best for `equals` and `hashCode`, sometimes you only have the entity identifier that provides a unique constraint.
>
> It's possible to use the entity identifier for equality check, but it needs a workaround:
>
> - you need to provide a constant value for `hashCode` so that the hash code value does not change before and after the entity is flushed.
> - you need to compare the entity identifier equality only for non-transient entities.

For details on mapping the identifier, see the Identifiers chapter.

2.5.8. Mapping optimistic locking

JPA defines support for optimistic locking based on either a version (sequential numeric) or timestamp strategy. To enable this style of optimistic locking simply add the `javax.persistence.Version` to the persistent attribute that defines the optimistic locking value. According to JPA, the valid types for these attributes are limited to:

- `int` or `Integer`
- `short` or `Short`
- `long` or `Long`
- `java.sql.Timestamp`

Example 111. `@Version` *annotation mapping*

JAVA

```
@Entity
public class Course {

    @Id
    private Integer id;

    @Version
    private Integer version;
    ...
}
```

JAVA

```
@Entity
public class Thing {

    @Id
    private Integer id;

    @Version
    private Timestamp ts;

    ...
}
```

JAVA

```
@Entity
public class Thing2 {

    @Id
    private Integer id;

    @Version
    private Instant ts;
    ...
}
```

Versionless optimistic locking

Although the default `@Version` property optimistic locking mechanism is sufficient in many situations, sometimes, you need rely on the actual database row column values to prevent **lost updates**.

Hibernate supports a form of optimistic locking that does not require a dedicated "version attribute". This is also useful for use with modeling legacy schemas.

The idea is that you can get Hibernate to perform "version checks" using either all of the entity's attributes, or just the attributes that have changed. This is achieved through the use of the `@OptimisticLocking` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLocking.html) annotation which defines a single attribute of type `org.hibernate.annotations.OptimisticLockType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLockType.html).

There are 4 available OptimisticLockTypes:

`NONE`
: optimistic locking is disabled even if there is a `@Version` annotation present

`VERSION` (the default)
: performs optimistic locking based on a `@Version` as described above

`ALL`

performs optimistic locking based on *all* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

`DIRTY`

performs optimistic locking based on *dirty* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

Versionless optimistic locking using `OptimisticLockType.ALL`

Example 112. `OptimisticLockType.ALL` *mapping example*

JAVA

```
@Entity(name = "Person")
@OptimisticLocking(type = OptimisticLockType.ALL)
@DynamicUpdate
public static class Person {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    private String country;

    private String city;

    @Column(name = "created_on")
    private Timestamp createdOn;

    //Getters and setters are omitted for brevity
}
```

When you need to modify the `Person` entity above:

Example 113. `OptimisticLockType.ALL` *update example*

JAVA

```
Person person = entityManager.find( Person.class, 1L );
person.setCity( "Washington D.C." );
```

```
SQL
UPDATE
    Person
SET
    city=?
WHERE
    id=?
    AND city=?
    AND country=?
    AND created_on=?
    AND "name"=?

-- binding parameter [1] as [VARCHAR] - [Washington D.C.]
-- binding parameter [2] as [BIGINT] - [1]
-- binding parameter [3] as [VARCHAR] - [New York]
-- binding parameter [4] as [VARCHAR] - [US]
-- binding parameter [5] as [TIMESTAMP] - [2016-11-16 16:05:12.876]
-- binding parameter [6] as [VARCHAR] - [John Doe]
```

As you can see, all the columns of the associated database row are used in the `WHERE` clause. If any column has changed after the row was loaded, there won't be any match, and a `StaleStateException` or an `OptimisticLockException` is going to be thrown.

> When using `OptimisticLockType.ALL`, you should also use `@DynamicUpdate` because the `UPDATE` statement must take into consideration all the entity property values.

Versionless optimistic locking using `OptimisticLockType.DIRTY`

The `OptimisticLockType.DIRTY` differs from `OptimisticLockType.ALL` in that it only takes into consideration the entity properties that have changed since the entity was loaded in the currently running Persistence Context.

Example 114. `OptimisticLockType.DIRTY` *mapping example*

```java
@Entity(name = "Person")
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@DynamicUpdate
@SelectBeforeUpdate
public static class Person {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    private String country;

    private String city;

    @Column(name = "created_on")
    private Timestamp createdOn;

    //Getters and setters are omitted for brevity
}
```

When you need to modify the `Person` entity above:

Example 115. `OptimisticLockType.DIRTY` *update example*

```java
Person person = entityManager.find( Person.class, 1L );
person.setCity( "Washington D.C." );
```

```sql
UPDATE
    Person
SET
    city=?
WHERE
    id=?
    and city=?

-- binding parameter [1] as [VARCHAR] - [Washington D.C.]
-- binding parameter [2] as [BIGINT] - [1]
-- binding parameter [3] as [VARCHAR] - [New York]
```

This time, only the database column that has changed was used in the `WHERE` clause.

> The main advantage of `OptimisticLockType.DIRTY` over `OptimisticLockType.ALL` and the default `OptimisticLockType.VERSION` used implicitly along with the `@Version` mapping, is that it allows you to minimize the risk of `OptimisticLockException` across non-overlapping entity property changes.

When using `OptimisticLockType.DIRTY`, you should also use `@DynamicUpdate` because the `UPDATE` statement must take into consideration all the dirty entity property values, and also the `@SelectBeforeUpdate` annotation so that detached entities are properly handled by the `Session#update(entity)` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html#update-java.lang.Object-) operation.

2.5.9. Access strategies

As a JPA provider, Hibernate can introspect both the entity attributes (instance fields) or the accessors (instance properties). By default, the placement of the `@Id` annotation gives the default access strategy. When placed on a field, Hibernate will assume field-based access. Place on the identifier getter, Hibernate will use property-based access.

> You should pay attention to Java Beans specification (https://docs.oracle.com/javase/7/docs/api/java/beans/Introspector.html#decapitalize(java.lang.String)) in regard to naming properties to avoid issues such as Property name beginning with at least two uppercase characters has odd functionality in HQL (https://hibernate.atlassian.net/browse/HCANN-63)!

Embeddable types inherit the access strategy from their parent entities.

Field-based access

Example 116. Field-based access

```java
@Entity
public class Simple {

    @Id
    private Integer id;

    public Integer getId() {
        return id;
    }

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

When using field-based access, adding other entity-level methods is much more flexible because Hibernate won't consider those part of the persistence state. To exclude a field from being part of the entity persistent state, the field must be marked with the `@Transient` annotation.

> Another advantage of using field-based access is that some entity attributes can be hidden from outside the entity. An example of such attribute is the entity `@Version` field, which must not be manipulated by the data access layer. With field-based access, we can simply omit the getter and the setter for this version field, and Hibernate can still leverage the optimistic concurrency control mechanism.

Property-based access

Example 117. Property-based access

```java
@Entity
public class Simple {

    private Integer id;

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

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

When using property-based access, Hibernate uses the accessors for both reading and writing the entity state. Every other method that will be added to the entity (e.g. helper methods for synchronizing both ends of a bidirectional one-to-many association) will have to be marked with the `@Transient` annotation.

Overriding the default access strategy

The default access strategy mechanism can be overridden with the JPA `@Access` annotation. In the following example, the `@Version` attribute is accessed by its field and not by its getter, like the rest of entity attributes.

Example 118. Overriding access strategy

```java
@Entity
public class Simple {

    private Integer id;

    @Version
    @Access( AccessType.FIELD )
    private Integer version;

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

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

Embeddable types and access strategy

Because embeddables are managed by their owning entities, the access strategy is therefore inherited from the entity too. This applies to both simple embeddable types as well as for collection of embeddables.

The embeddable types can overrule the default implicit access strategy (inherited from the owning entity). In the following example, the embeddable uses property-based access, no matter what access strategy the owning entity is choosing:

Example 119. Embeddable with exclusive access strategy

JAVA

```
@Embeddable
@Access(AccessType.PROPERTY)
public static class Change {

    private String path;

    private String diff;

    public Change() {}

    @Column(name = "path", nullable = false)
    public String getPath() {
        return path;
    }

    public void setPath(String path) {
        this.path = path;
    }

    @Column(name = "diff", nullable = false)
    public String getDiff() {
        return diff;
    }

    public void setDiff(String diff) {
        this.diff = diff;
    }
}
```

The owning entity can use field-based access, while the embeddable uses property-based access as it has chosen explicitly:

Example 120. Entity including a single embeddable type

```java
@Entity
public class Patch {

    @Id
    private Long id;

    @Embedded
    private Change change;
}
```

This works also for collection of embeddable types:

Example 121. Entity including a collection of embeddable types

```java
@Entity
public class Patch {

    @Id
    private Long id;

    @ElementCollection
    @CollectionTable(
        name="patch_change",
        joinColumns=@JoinColumn(name="patch_id")
    )
    @OrderColumn(name = "index_id")
    private List<Change> changes = new ArrayList<>();

    public List<Change> getChanges() {
        return changes;
    }
}
```

2.6. Identifiers

Identifiers model the primary key of an entity. They are used to uniquely identify each specific entity.

Hibernate and JPA both make the following assumptions about the corresponding database column(s):

`UNIQUE`

The values must uniquely identify each row.

`NOT NULL`

The values cannot be null. For composite ids, no part can be null.

`IMMUTABLE`

The values, once inserted, can never be changed. This is more a general guide, than a hard-fast rule as opinions vary. JPA defines the behavior of changing the value of the identifier attribute to be undefined; Hibernate simply does not support that. In cases where the values for the PK you have chosen will be updated, Hibernate recommends mapping the mutable value as a natural id, and use a surrogate id for the PK. See Natural Ids.

> Technically the identifier does not have to map to the column(s) physically defined as the table primary key. They just need to map to column(s) that uniquely identify each row. However this documentation will continue to use the terms identifier and primary key interchangeably.

Every entity must define an identifier. For entity inheritance hierarchies, the identifier must be defined just on the entity that is the root of the hierarchy.

An identifier might be simple (single value) or composite (multiple values).

2.6.1. Simple identifiers

Simple identifiers map to a single basic attribute, and are denoted using the `javax.persistence.Id` annotation.

According to JPA only the following types should be used as identifier attribute types:

- any Java primitive type
- any primitive wrapper type
- `java.lang.String`
- `java.util.Date` (TemporalType#DATE)
- `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

Any types used for identifier attributes beyond this list will not be portable.

Assigned identifiers

Values for simple identifiers can be assigned, as we have seen in the examples above. The expectation for assigned identifier values is that the application assigns (sets them on the entity attribute) prior to calling save/persist.

Example 122. Simple assigned identifier

```java
@Entity
public class MyEntity {

    @Id
    public Integer id;

    ...
}
```

Generated identifiers

Values for simple identifiers can be generated. To denote that an identifier attribute is generated, it is annotated with `javax.persistence.GeneratedValue`

Example 123. Simple generated identifier

```java
@Entity
public class MyEntity {

    @Id
    @GeneratedValue
    public Integer id;

    ...
}
```

Additionally, to the type restriction list above, JPA says that if using generated identifier values (see below) only integer types (short, int, long) will be portably supported.

The expectation for generated identifier values is that Hibernate will generate the value when the save/persist occurs.

Identifier value generations strategies are discussed in detail in the Generated identifier values section.

2.6.2. Composite identifiers

Composite identifiers correspond to one or more persistent attributes. Here are the rules governing composite identifiers, as defined by the JPA specification.

- The composite identifier must be represented by a "primary key class". The primary key class may be defined using the `javax.persistence.EmbeddedId` annotation (see Composite identifiers with `@EmbeddedId`), or defined using the `javax.persistence.IdClass` annotation (see Composite identifiers with `@IdClass`).
- The primary key class must be public and must have a public no-arg constructor.
- The primary key class must be serializable.
- The primary key class must define equals and hashCode methods, consistent with equality for the underlying database types to which the primary key is mapped.

> The restriction that a composite identifier has to be represented by a "primary key class" is only JPA specific. Hibernate does allow composite identifiers to be defined without a "primary key class", although that modeling technique is deprecated and therefore omitted from this discussion.

The attributes making up the composition can be either basic, composite, ManyToOne. Note especially that collections and one-to-ones are never appropriate.

2.6.3. Composite identifiers with `@EmbeddedId`

Modeling a composite identifier using an EmbeddedId simply means defining an embeddable to be a composition for the one or more attributes making up the identifier, and then exposing an attribute of that embeddable type on the entity.

Example 124. Basic EmbeddedId

```java
@Entity
public class Login {

    @EmbeddedId
    private PK pk;

    @Embeddable
    public static class PK implements Serializable {

        private String system;

        private String username;

        ...
    }

    ...
}
```

As mentioned before, EmbeddedIds can even contain ManyToOne attributes.

Example 125. EmbeddedId with ManyToOne

```java
@Entity
public class Login {

    @EmbeddedId
    private PK pk;

    @Embeddable
    public static class PK implements Serializable {

        @ManyToOne
        private System system;

        private String username;

        ...
    }

    ...
}
```

> Hibernate supports directly modeling the ManyToOne in the PK class, whether EmbeddedId or IdClass. However that is not portably supported by the JPA specification. In JPA terms one would use "derived identifiers"; for details, see Derived Identifiers.

2.6.4. Composite identifiers with `@IdClass`

Modeling a composite identifier using an IdClass differs from using an EmbeddedId in that the entity defines each individual attribute making up the composition. The IdClass simply acts as a "shadow".

Example 126. Basic IdClass

JAVA

```
@Entity
@IdClass( PK.class )
public class Login {

    @Id
    private String system;

    @Id
    private String username;

    public static class PK implements Serializable {

        private String system;

        private String username;

        ...
    }

    ...
}
```

Non-aggregated composite identifiers can also contain ManyToOne attributes as we saw with aggregated ones (still non-portably).

Example 127. IdClass with ManyToOne

```java
@Entity
@IdClass( PK.class )
public class Login {

    @Id
    @ManyToOne
    private System system;

    @Id
    private String username;

    public static class PK implements Serializable {

        private System system;

        private String username;

        ...
    }

    ...
}
```

With non-aggregated composite identifiers, Hibernate also supports "partial" generation of the composite values.

Example 128. IdClass with partial generation

```java
@Entity
@IdClass( PK.class )
public class LogFile {

    @Id
    private String name;

    @Id
    private LocalDate date;

    @Id
    @GeneratedValue
    private Integer uniqueStamp;

    public static class PK implements Serializable {

        private String name;

        private LocalDate date;

        private Integer uniqueStamp;

        ...
    }

    ...
}
```

> This feature exists because of a highly questionable interpretation of the JPA specification made by the SpecJ committee. Hibernate does not feel that JPA defines support for this, but added the feature simply to be usable in SpecJ benchmarks. Use of this feature may or may not be portable from a JPA perspective.

2.6.5. Composite identifiers with associations

Hibernate allows defining a composite identifier out of entity associations. In the following example, the `PersonAddress` entity identifier is formed of two `@ManyToOne` associations.

Example 129. Composite identifiers with associations

JAVA

```
@Entity
public class PersonAddress  implements Serializable {

    @Id
    @ManyToOne
    private Person person;

    @Id
    @ManyToOne()
    private Address address;

    public PersonAddress() {}

    public PersonAddress(Person person, Address address) {
        this.person = person;
        this.address = address;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        PersonAddress that = (PersonAddress) o;
        return Objects.equals(person, that.person) &&
                Objects.equals(address, that.address);
    }

    @Override
    public int hashCode() {
        return Objects.hash(person, address);
    }
}

@Entity
public class Person  {

    @Id
    @GeneratedValue
    private Long id;
```

```
    @NaturalId
    private String registrationNumber;

    public Person() {}

    public Person(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public Long getId() {
        return id;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(registrationNumber, person.registrationNumber);
    }

    @Override
    public int hashCode() {
        return Objects.hash(registrationNumber);
    }
}

@Entity
public class Address  {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    private String number;

    private String postalCode;

    public Address() {}

    public Address(String street, String number, String postalCode) {
        this.street = street;
        this.number = number;
        this.postalCode = postalCode;
    }

    public Long getId() {
        return id;
    }

    public String getStreet() {
        return street;
    }

    public String getNumber() {
        return number;
```

```
    }

    public String getPostalCode() {
        return postalCode;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Address address = (Address) o;
        return Objects.equals(street, address.street) &&
                Objects.equals(number, address.number) &&
                Objects.equals(postalCode, address.postalCode);
    }

    @Override
    public int hashCode() {
        return Objects.hash(street, number, postalCode);
    }
}
```

Although the mapping is much simpler than using an `@EmbeddedId` or an `@IdClass`, there's no separation between the entity instance and the actual identifier. To query this entity, an instance of the entity itself must be supplied to the persistence context.

Example 130. Composite identifiers with associations query

JAVA

```
PersonAddress personAddress = entityManager.find(
    PersonAddress.class,
    new PersonAddress( person, address )
);
```

2.6.6. Generated identifier values

> For discussion of generated values for non-identifier attributes, see Generated properties.

Hibernate supports identifier value generation across a number of different types. Remember that JPA portably defines identifier value generation just for integer types.

Identifier value generation is indicates using the `javax.persistence.GeneratedValue` annotation. The most important piece of information here is the specified `javax.persistence.GenerationType` which indicates how values will be generated.

> The discussions below assume that the application is using Hibernate's "new generator mappings" as indicated by the `hibernate.id.new_generator_mappings` setting or `MetadataBuilder.enableNewIdentifierGeneratorSupport` method during bootstrap. Starting with Hibernate 5, this is set to true by default. If applications set this to false the resolutions discussed here will be very different. The rest of the discussion here assumes this setting is enabled (true).

`AUTO` (the default)
: Indicates that the persistence provider (Hibernate) should choose an appropriate generation strategy. See Interpreting AUTO.

`IDENTITY`
: Indicates that database IDENTITY columns will be used for primary key value generation. See Using IDENTITY columns.

`SEQUENCE`
: Indicates that database sequence should be used for obtaining primary key values. See Using sequences.

`TABLE`
: Indicates that a database table should be used for obtaining primary key values. See Using identifier table.

2.6.7. Interpreting AUTO

How a persistence provider interprets the AUTO generation type is left up to the provider.

The default behavior is to look at the java type of the identifier attribute.

If the identifier type is UUID, Hibernate is going to use an UUID identifier.

If the identifier type is numerical (e.g. `Long` , `Integer`), then Hibernate is going to use the `IdGeneratorStrategyInterpreter` to resolve the identifier generator strategy. The `IdGeneratorStrategyInterpreter` has two implementations:

`FallbackInterpreter`

This is the default strategy since Hibernate 5.0. For older versions, this strategy is enabled through the `hibernate.id.new_generator_mappings` configuration property . When using this strategy, `AUTO` always resolves to `SequenceStyleGenerator` . If the underlying database supports sequences, then a SEQUENCE generator is used. Otherwise, a TABLE generator is going to be used instead.

`LegacyFallbackInterpreter`

This is a legacy mechanism that was used by Hibernate prior to version 5.0 or when the `hibernate.id.new_generator_mappings` configuration property is false. The legacy strategy maps `AUTO` to the `native` generator strategy which uses the Dialect#getNativeIdentifierGeneratorStrategy (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/dialect/Dialect.html#getNativeIdentifierGeneratorStrategy--) to resolve the actual identifier generator (e.g. `identity` or `sequence`).

2.6.8. Using sequences

For implementing database sequence-based identifier value generation Hibernate makes use of its `org.hibernate.id.enhanced.SequenceStyleGenerator` id generator. It is important to note that SequenceStyleGenerator is capable of working against databases that do not support sequences by switching to a table as the underlying backing. This gives Hibernate a huge degree of portability across databases while still maintaining consistent id generation behavior (versus say choosing between SEQUENCE and IDENTITY). This backing storage is completely transparent to the user.

The preferred (and portable) way to configure this generator is using the JPA-defined `javax.persistence.SequenceGenerator` annotation.

The simplest form is to simply request sequence generation; Hibernate will use a single, implicitly-named sequence (`hibernate_sequence`) for all such unnamed definitions.

Example 131. Unnamed sequence

JAVA

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue( generation = SEQUENCE )
    public Integer id;

    ...
}
```

Or a specifically named sequence can be requested.

Example 132. Named sequence

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue( generation = SEQUENCE, name = "my_sequence" )
    public Integer id;

    ...
}
```

Use `javax.persistence.SequenceGenerator` to specify additional configuration.

Example 133. Configured sequence

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue( generation = SEQUENCE, name = "my_sequence" )
    @SequenceGenerator( name = "my_sequence", schema = "globals", allocationSize = 30 )
    public Integer id;

    ...
}
```

2.6.9. Using IDENTITY columns

For implementing identifier value generation based on IDENTITY columns, Hibernate makes use of its `org.hibernate.id.IdentityGenerator` id generator which expects the identifier to generated by INSERT into the table. IdentityGenerator understands 3 different ways that the INSERT-generated value might be retrieved:

- If Hibernate believes the JDBC environment supports `java.sql.Statement#getGeneratedKeys`, then that approach will be used for extracting the IDENTITY generated keys.
- Otherwise, if `Dialect#supportsInsertSelectIdentity` reports true, Hibernate will use the Dialect specific INSERT+SELECT statement syntax.
- Otherwise, Hibernate will expect that the database supports some form of asking for the most recently inserted IDENTITY value via a separate SQL command as indicated by `Dialect#getIdentitySelectString`.

> It is important to realize that this imposes a runtime behavior where the entity row **must** be physically inserted prior to the identifier value being known. This can mess up extended persistence contexts (conversations). Because of the runtime imposition/inconsistency Hibernate suggest other forms of identifier value generation be used.

> There is yet another important runtime impact of choosing IDENTITY generation: Hibernate will not be able to JDBC batching for inserts of the entities that use IDENTITY generation. The importance of this depends on the application specific use cases. If the application is not usually creating many new instances of a given type of entity that uses IDENTITY generation, then this is not an important impact since batching would not have been helpful anyway.

2.6.10. Using identifier table

Hibernate achieves table-based identifier generation based on its `org.hibernate.id.enhanced.TableGenerator` id generator which defines a table capable of holding multiple named value segments for any number of entities.

Example 134. Table generator table structure

```
SQL
create table hibernate_sequences(
    sequence_name VARCHAR NOT NULL,
    next_val INTEGER NOT NULL
)
```

The basic idea is that a given table-generator table (`hibernate_sequences` for example) can hold multiple segments of identifier generation values.

Example 135. Unnamed table generator

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue( generation = TABLE )
    public Integer id;

    ...
}
```

JAVA

If no table name is given Hibernate assumes an implicit name of `hibernate_sequences`. Additionally, because no `javax.persistence.TableGenerator#pkColumnValue` is specified, Hibernate will use the default segment (`sequence_name='default'`) from the hibernate_sequences table.

2.6.11. Using UUID generation

As mentioned above, Hibernate supports UUID identifier value generation. This is supported through its `org.hibernate.id.UUIDGenerator` id generator.

`UUIDGenerator` supports pluggable strategies for exactly how the UUID is generated. These strategies are defined by the `org.hibernate.id.UUIDGenerationStrategy` contract. The default strategy is a version 4 (random) strategy according to IETF RFC 4122. Hibernate does ship with an alternative strategy which is a RFC 4122 version 1 (time-based) strategy (using ip address rather than mac address).

Example 136. Implicitly using the random UUID strategy

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue
    public UUID id;

    ...
}
```

JAVA

To specify an alternative generation strategy, we'd have to define some configuration via `@GenericGenerator`. Here we choose the RFC 4122 version 1 compliant strategy named `org.hibernate.id.uuid.CustomVersionOneStrategy`.

Example 137. Implicitly using the random UUID strategy

JAVA

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue( generator = "uuid" )
    @GenericGenerator(
        name = "uuid",
        strategy = "org.hibernate.id.UUIDGenerator",
        parameters = {
            @Parameter(
                name = "uuid_gen_strategy_class",
                value = "org.hibernate.id.uuid.CustomVersionOneStrategy"
            )
        }
    )
    public UUID id;

    ...

}
```

2.6.12. Optimizers

Most of the Hibernate generators that separately obtain identifier values from database structures support the use of pluggable optimizers. Optimizers help manage the number of times Hibernate has to talk to the database in order to generate identifier values. For example, with no optimizer applied to a sequence-generator, every time the application asked Hibernate to generate an identifier it would need to grab the next sequence value from the database. But if we can minimize the number of times we need to communicate with the database here, the application will be able to perform better. Which is, in fact, the role of these optimizers.

none

No optimization is performed. We communicate with the database each and every time an identifier value is needed from the generator.

pooled-lo

The pooled-lo optimizer works on the principle that the increment-value is encoded into the database table/sequence structure. In sequence-terms this means that the sequence is defined with a greater-that-1 increment size.

For example, consider a brand new sequence defined as `create sequence m_sequence start with 1 increment by 20`. This sequence essentially defines a "pool" of 20 usable id values each and every time we ask it for its next-value. The pooled-lo optimizer interprets the next-value as the low end of that pool.

So when we first ask it for next-value, we'd get 1. We then assume that the valid pool would be the values from 1-20 inclusive.

The next call to the sequence would result in 21, which would define 21-40 as the valid range. And so on. The "lo" part of the name indicates that the value from the database table/sequence is interpreted as the pool lo(w) end.

pooled

Just like pooled-lo, except that here the value from the table/sequence is interpreted as the high end of the value pool.

hilo; legacy-hilo

Define a custom algorithm for generating pools of values based on a single value from a table or sequence.

These optimizers are not recommended for use. They are maintained (and mentioned) here simply for use by legacy applications that used these strategies previously.

> Applications can also implement and use their own optimizer strategies, as defined by the `org.hibernate.id.enhanced.Optimizer` contract.

2.6.13. Using `@GenericGenerator`

`@GenericGenerator` allows integration of any Hibernate `org.hibernate.id.IdentifierGenerator` implementation, including any of the specific ones discussed here and any custom ones.

To make use of the pooled or pooled-lo optimizers, the entity mapping must use the `@GenericGenerator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenericGenerator.html) annotation:

Example 138. Pooled-lo optimizer mapping using `@GenericGenerator` mapping

```java
@Entity(name = "Product")
public static class Product {

    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "product_generator"
    )
    @GenericGenerator(
        name = "product_generator",
        strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
        parameters = {
            @Parameter(name = "sequence_name", value = "product_sequence"),
            @Parameter(name = "initial_value", value = "1"),
            @Parameter(name = "increment_size", value = "3"),
            @Parameter(name = "optimizer", value = "pooled-lo")
        }
    )
    private Long id;

    @Column(name = "p_name")
    private String name;

    @Column(name = "p_number")
    private String number;

    //Getters and setters are omitted for brevity

}
```

Now, when saving 5 `Person` entities and flushing the Persistence Context after every 3 entities:

Example 139. Pooled-lo optimizer mapping using `@GenericGenerator` mapping

```java
for ( long i = 1; i <= 5; i++ ) {
    if(i % 3 == 0) {
        entityManager.flush();
    }
    Product product = new Product();
    product.setName( String.format( "Product %d", i ) );
    product.setNumber( String.format( "P_100_%d", i ) );
    entityManager.persist( product );
}
```

SQL

```
CALL NEXT VALUE FOR product_sequence

INSERT INTO Product (p_name, p_number, id)
VALUES (?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Product 1]
-- binding parameter [2] as [VARCHAR] - [P_100_1]
-- binding parameter [3] as [BIGINT]  - [1]

INSERT INTO Product (p_name, p_number, id)
VALUES (?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Product 2]
-- binding parameter [2] as [VARCHAR] - [P_100_2]
-- binding parameter [3] as [BIGINT]  - [2]

CALL NEXT VALUE FOR product_sequence

INSERT INTO Product (p_name, p_number, id)
VALUES (?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Product 3]
-- binding parameter [2] as [VARCHAR] - [P_100_3]
-- binding parameter [3] as [BIGINT]  - [3]

INSERT INTO Product (p_name, p_number, id)
VALUES (?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Product 4]
-- binding parameter [2] as [VARCHAR] - [P_100_4]
-- binding parameter [3] as [BIGINT]  - [4]

INSERT INTO Product (p_name, p_number, id)
VALUES (?, ?, ?)

-- binding parameter [1] as [VARCHAR] - [Product 5]
-- binding parameter [2] as [VARCHAR] - [P_100_5]
-- binding parameter [3] as [BIGINT]  - [5]
```

As you can see from the list of generated SQL statements, you can insert 3 entities for one database sequence call. This way, the pooled and the pooled-lo optimizers allow you to reduce the number of database roundtrips, therefore reducing the overall transaction response time.

2.6.14. Derived Identifiers

JPA 2.0 added support for derived identifiers which allow an entity to borrow the identifier from a many-to-one or one-to-one association.

Example 140. Derived identifier with `@MapsId`

JAVA

```
@Entity
public class Person  {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    public Person() {}

    public Person(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public Long getId() {
        return id;
    }

    public String getRegistrationNumber() {
        return registrationNumber;
    }
}

@Entity
public class PersonDetails  {

    @Id
    private Long id;

    private String nickName;

    @ManyToOne
    @MapsId
    private Person person;

    public String getNickName() {
        return nickName;
    }

    public void setNickName(String nickName) {
        this.nickName = nickName;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

In the example above, the `PersonDetails` entity uses the `id` column for both the entity identifier and for the many-to-one association to the `Person` entity. The value of the `PersonDetails` entity identifier is "derived" from the identifier of its parent `Person` entity. The `@MapsId` annotation can also reference columns from an `@EmbeddedId` identifier as well.

The previous example can also be mapped using `@PrimaryKeyJoinColumn`.

Example 141. Derived identifier `@PrimaryKeyJoinColumn`

JAVA

```
@Entity
public class PersonDetails  {

    @Id
    private Long id;

    private String nickName;

    @ManyToOne
    @PrimaryKeyJoinColumn
    private Person person;

    public String getNickName() {
        return nickName;
    }

    public void setNickName(String nickName) {
        this.nickName = nickName;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
        this.id = person.getId();
    }
}
```

> Unlike `@MapsId`, the application developer is responsible for ensuring that the identifier and the many-to-one (or one-to-one) association are in sync.

2.7. Associations

Associations describe how two or more entities form a relationship based on a database joining semantics.

2.7.1. `@ManyToOne`

`@ManyToOne` is the most common association, having a direct equivalent in the relational database as well (e.g. foreign key), and so it establishes a relationship between a child entity and a parent.

Example 142. `@ManyToOne` *association*

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    public Person() {
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne
    @JoinColumn(name = "person_id",
            foreignKey = @ForeignKey(name = "PERSON_ID_FK")
    )
    private Person person;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getNumber() {
        return number;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

```
SQL
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
 )

ALTER TABLE Phone
ADD CONSTRAINT PERSON_ID_FK
FOREIGN KEY (person_id) REFERENCES Person
```

Each entity has a lifecycle of its own. Once the `@ManyToOne` association is set, Hibernate will set the associated database foreign key column.

Example 143. `@ManyToOne` *association lifecycle*

```
JAVA
Person person = new Person();
entityManager.persist( person );

Phone phone = new Phone( "123-456-7890" );
phone.setPerson( person );
entityManager.persist( phone );

entityManager.flush();
phone.setPerson( null );
```

```
SQL
INSERT INTO Person ( id )
VALUES ( 1 )

INSERT INTO Phone ( number, person_id, id )
VALUES ( '123-456-7890', 1, 2 )

UPDATE Phone
SET    number = '123-456-7890',
       person_id = NULL
WHERE  id = 2
```

2.7.2. `@OneToMany`

The `@OneToMany` association links a parent entity with one or more child entities. If the `@OneToMany` doesn't have a mirroring `@ManyToOne` association on the child side, the `@OneToMany` association is unidirectional. If there is a `@ManyToOne` association on the child side, the `@OneToMany` association is bidirectional and the application developer can navigate this relationship from both

ends.

Unidirectional `@OneToMany`

When using a unidirectional `@OneToMany` association, Hibernate resorts to using a link table between the two joining entities.

Example 144. Unidirectional `@OneToMany` association

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    public Person() {
    }

    public List<Phone> getPhones() {
        return phones;
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getNumber() {
        return number;
    }
}
```

```sql
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person_Phone (
    Person_id BIGINT NOT NULL ,
    phones_id BIGINT NOT NULL
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    PRIMARY KEY ( id )
)

ALTER TABLE Person_Phone
ADD CONSTRAINT UK_9uhc5itwc9h5gcng944pcaslf
UNIQUE (phones_id)

ALTER TABLE Person_Phone
ADD CONSTRAINT FKr38us2n8g5p9rj0b494sd3391
FOREIGN KEY (phones_id) REFERENCES Phone

ALTER TABLE Person_Phone
ADD CONSTRAINT FK2ex4e4p7w1cj310kg2woisjl2
FOREIGN KEY (Person_id) REFERENCES Person
```

> The `@OneToMany` association is by definition a parent association, even if it's a unidirectional or a bidirectional one. Only the parent side of an association makes sense to cascade its entity state transitions to children.

Example 145. Cascading `@OneToMany` association

JAVA

```
Person person = new Person();
Phone phone1 = new Phone( "123-456-7890" );
Phone phone2 = new Phone( "321-654-0987" );

person.getPhones().add( phone1 );
person.getPhones().add( phone2 );
entityManager.persist( person );
entityManager.flush();

person.getPhones().remove( phone1 );
```

SQL

```
INSERT INTO Person
       ( id )
VALUES ( 1 )

INSERT INTO Phone
       ( number, id )
VALUES ( '123 - 456 - 7890', 2 )

INSERT INTO Phone
       ( number, id )
VALUES ( '321 - 654 - 0987', 3 )

INSERT INTO Person_Phone
       ( Person_id, phones_id )
VALUES ( 1, 2 )

INSERT INTO Person_Phone
       ( Person_id, phones_id )
VALUES ( 1, 3 )

DELETE FROM Person_Phone
WHERE  Person_id = 1

INSERT INTO Person_Phone
       ( Person_id, phones_id )
VALUES ( 1, 3 )

DELETE FROM Phone
WHERE  id = 2
```

When persisting the `Person` entity, the cascade will propagate the persist operation to the underlying `Phone` children as well. Upon removing a `Phone` from the phones collection, the association row is deleted from the link table, and the `orphanRemoval` attribute will trigger a `Phone` removal as well.

> The unidirectional associations are not very efficient when it comes to removing child entities. In this particular example, upon flushing the persistence context, Hibernate deletes all database child entries and reinserts the ones that are still found in the in-memory persistence context.
>
> On the other hand, a bidirectional `@OneToMany` association is much more efficient because the child entity controls the association.

Bidirectional `@OneToMany`

The bidirectional `@OneToMany` association also requires a `@ManyToOne` association on the child side. Although the Domain Model exposes two sides to navigate this association, behind the scenes, the relational database has only one foreign key for this relationship.

Every bidirectional association must have one owning side only (the child side), the other one being referred to as the *inverse* (or the `mappedBy`) side.

Example 146. `@OneToMany` *association mappedBy the* `@ManyToOne` *side*

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public List<Phone> getPhones() {
        return phones;
    }

    public void addPhone(Phone phone) {
        phones.add( phone );
        phone.setPerson( this );
    }

    public void removePhone(Phone phone) {
        phones.remove( phone );
        phone.setPerson( null );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    @Column(name = "`number`", unique = true)
    private String number;

    @ManyToOne
    private Person person;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    public Long getId() {
        return id;
    }
```

```
    public String getNumber() {
        return number;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

SQL

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
)

ALTER TABLE Phone
ADD CONSTRAINT UK_l329ab0g4c1t78onljnxmbnp6
UNIQUE (number)

ALTER TABLE Phone
ADD CONSTRAINT FKmw13yfsjypiiq0i1osdkaeqpg
FOREIGN KEY (person_id) REFERENCES Person
```

> Whenever a bidirectional association is formed, the application developer must make sure both sides are in-sync at all times. The `addPhone()` and `removePhone()` are utilities methods that synchronize both ends whenever a child element is added or removed.

Because the `Phone` class has a `@NaturalId` column (the phone number being unique), the `equals()` and the `hashCode()` can make use of this property, and so the `removePhone()` logic is reduced to the `remove()` Java `Collection` method.

Example 147. Bidirectional `@OneToMany` with an owner `@ManyToOne` side lifecycle

JAVA

```
Person person = new Person();
Phone phone1 = new Phone( "123-456-7890" );
Phone phone2 = new Phone( "321-654-0987" );

person.addPhone( phone1 );
person.addPhone( phone2 );
entityManager.persist( person );
entityManager.flush();

person.removePhone( phone1 );
```

SQL

```
INSERT INTO Phone
       ( number, person_id, id )
VALUES ( '123-456-7890', NULL, 2 )

INSERT INTO Phone
       ( number, person_id, id )
VALUES ( '321-654-0987', NULL, 3 )

DELETE FROM Phone
WHERE  id = 2
```

Unlike the unidirectional `@OneToMany`, the bidirectional association is much more efficient when managing the collection persistence state. Every element removal only requires a single update (in which the foreign key column is set to `NULL`), and, if the child entity lifecycle is bound to its owning parent so that the child cannot exist without its parent, then we can annotate the association with the `orphan-removal` attribute and disassociating the child will trigger a delete statement on the actual child table row as well.

2.7.3. `@OneToOne`

The `@OneToOne` association can either be unidirectional or bidirectional. A unidirectional association follows the relational database foreign key semantics, the client-side owning the relationship. A bidirectional association features a `mappedBy` `@OneToOne` parent side too.

Unidirectional `@OneToOne`

Example 148. Unidirectional `@OneToOne`

JAVA

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne
    @JoinColumn(name = "details_id")
    private PhoneDetails details;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getNumber() {
        return number;
    }

    public PhoneDetails getDetails() {
        return details;
    }

    public void setDetails(PhoneDetails details) {
        this.details = details;
    }
}

@Entity(name = "PhoneDetails")
public static class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;

    public PhoneDetails() {
    }

    public PhoneDetails(String provider, String technology) {
        this.provider = provider;
        this.technology = technology;
    }
```

```
    public String getProvider() {
        return provider;
    }

    public String getTechnology() {
        return technology;
    }

    public void setTechnology(String technology) {
        this.technology = technology;
    }
}
```

```SQL
CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    details_id BIGINT ,
    PRIMARY KEY ( id )
)

CREATE TABLE PhoneDetails (
    id BIGINT NOT NULL ,
    provider VARCHAR(255) ,
    technology VARCHAR(255) ,
    PRIMARY KEY ( id )
)

ALTER TABLE Phone
ADD CONSTRAINT FKnoj7cj83ppfqbnvqqa5kolub7
FOREIGN KEY (details_id) REFERENCES PhoneDetails
```

From a relational database point of view, the underlying schema is identical to the unidirectional `@ManyToOne` association, as the client-side controls the relationship based on the foreign key column.

But then, it's unusual to consider the `Phone` as a client-side and the `PhoneDetails` as the parent-side because the details cannot exist without an actual phone. A much more natural mapping would be if the `Phone` were the parent-side, therefore pushing the foreign key into the `PhoneDetails` table. This mapping requires a bidirectional `@OneToOne` association as you can see in the following example:

Bidirectional `@OneToOne`

Example 149. Bidirectional `@OneToOne`

JAVA

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne(mappedBy = "phone", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
    private PhoneDetails details;

    public Phone() {
    }

    public Phone(String number) {
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getNumber() {
        return number;
    }

    public PhoneDetails getDetails() {
        return details;
    }

    public void addDetails(PhoneDetails details) {
        details.setPhone( this );
        this.details = details;
    }

    public void removeDetails() {
        if ( details != null ) {
            details.setPhone( null );
            this.details = null;
        }
    }
}

@Entity(name = "PhoneDetails")
public static class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;
```

```
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;

    public PhoneDetails() {
    }

    public PhoneDetails(String provider, String technology) {
        this.provider = provider;
        this.technology = technology;
    }

    public String getProvider() {
        return provider;
    }

    public String getTechnology() {
        return technology;
    }

    public void setTechnology(String technology) {
        this.technology = technology;
    }

    public Phone getPhone() {
        return phone;
    }

    public void setPhone(Phone phone) {
        this.phone = phone;
    }
}
```

SQL

```
CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE PhoneDetails (
    id BIGINT NOT NULL ,
    provider VARCHAR(255) ,
    technology VARCHAR(255) ,
    phone_id BIGINT ,
    PRIMARY KEY ( id )
)

ALTER TABLE PhoneDetails
ADD CONSTRAINT FKeotuev8ja8v0sdh29dynqj05p
FOREIGN KEY (phone_id) REFERENCES Phone
```

This time, the `PhoneDetails` owns the association, and, like any bidirectional association, the parent-side can propagate its lifecycle to the child-side through cascading.

Example 150. Bidirectional `@OneToOne` lifecycle

JAVA

```
Phone phone = new Phone( "123-456-7890" );
PhoneDetails details = new PhoneDetails( "T-Mobile", "GSM" );

phone.addDetails( details );
entityManager.persist( phone );
```

SQL

```
INSERT INTO Phone ( number, id )
VALUES ( '123 - 456 - 7890', 1 )

INSERT INTO PhoneDetails ( phone_id, provider, technology, id )
VALUES ( 1, 'T - Mobile, GSM', 2 )
```

When using a bidirectional `@OneToOne` association, Hibernate enforces the unique constraint upon fetching the child-side. If there are more than one children associated with the same parent, Hibernate will throw a `org.hibernate.exception.ConstraintViolationException`. Continuing the previous example, when adding another `PhoneDetails`, Hibernate validates the uniqueness constraint when reloading the `Phone` object.

Example 151. Bidirectional `@OneToOne` unique constraint

JAVA

```
PhoneDetails otherDetails = new PhoneDetails( "T-Mobile", "CDMA" );
otherDetails.setPhone( phone );
entityManager.persist( otherDetails );
entityManager.flush();
entityManager.clear();

//throws javax.persistence.PersistenceException: org.hibernate.HibernateException: More than one row with the given
identifier was found: 1
phone = entityManager.find( Phone.class, phone.getId() );
```

2.7.4. `@ManyToMany`

The `@ManyToMany` association requires a link table that joins two entities. Like the `@OneToMany` association, `@ManyToMany` can be a either unidirectional or bidirectional.

Unidirectional `@ManyToMany`

Example 152. Unidirectional `@ManyToMany`

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();

    public Person() {
    }

    public List<Address> getAddresses() {
        return addresses;
    }
}

@Entity(name = "Address")
public static class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    @Column(name = "`number`")
    private String number;

    public Address() {
    }

    public Address(String street, String number) {
        this.street = street;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getStreet() {
        return street;
    }

    public String getNumber() {
        return number;
    }
}
```

```sql
CREATE TABLE Address (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    street VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person_Address (
    Person_id BIGINT NOT NULL ,
    addresses_id BIGINT NOT NULL
)

ALTER TABLE Person_Address
ADD CONSTRAINT FKm7j0bnabh2yr0pe99il1d066u
FOREIGN KEY (addresses_id) REFERENCES Address

ALTER TABLE Person_Address
ADD CONSTRAINT FKba7rc9qe2vh44u93u0p2auwti
FOREIGN KEY (Person_id) REFERENCES Person
```

Just like with unidirectional `@OneToMany` associations, the link table is controlled by the owning side.

When an entity is removed from the `@ManyToMany` collection, Hibernate simply deletes the joining record in the link table. Unfortunately, this operation requires removing all entries associated with a given parent and recreating the ones that are listed in the current running persistent context.

Example 153. Unidirectional `@ManyToMany` lifecycle

```java
Person person1 = new Person();
Person person2 = new Person();

Address address1 = new Address( "12th Avenue", "12A" );
Address address2 = new Address( "18th Avenue", "18B" );

person1.getAddresses().add( address1 );
person1.getAddresses().add( address2 );

person2.getAddresses().add( address1 );

entityManager.persist( person1 );
entityManager.persist( person2 );

entityManager.flush();

person1.getAddresses().remove( address1 );
```

SQL

```
INSERT INTO Person ( id )
VALUES ( 1 )

INSERT INTO Address ( number, street, id )
VALUES ( '12A', '12th Avenue', 2 )

INSERT INTO Address ( number, street, id )
VALUES ( '18B', '18th Avenue', 3 )

INSERT INTO Person ( id )
VALUES ( 4 )

INSERT INTO Person_Address ( Person_id, addresses_id )
VALUES ( 1, 2 )
INSERT INTO Person_Address ( Person_id, addresses_id )
VALUES ( 1, 3 )
INSERT INTO Person_Address ( Person_id, addresses_id )
VALUES ( 4, 2 )

DELETE FROM Person_Address
WHERE  Person_id = 1

INSERT INTO Person_Address ( Person_id, addresses_id )
VALUES ( 1, 3 )
```

> For `@ManyToMany` associations, the `REMOVE` entity state transition doesn't make sense to be cascaded because it will propagate beyond the link table. Since the other side might be referenced by other entities on the parent-side, the automatic removal might end up in a `ConstraintViolationException`.

For example, if `@ManyToMany(cascade = CascadeType.ALL)` was defined and the first person would be deleted, Hibernate would throw an exception because another person is still associated with the address that's being deleted.

JAVA

```
Person person1 = entityManager.find(Person.class, personId);
entityManager.remove(person1);

Caused by: javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException: could not
execute statement
Caused by: org.hibernate.exception.ConstraintViolationException: could not execute statement
Caused by: java.sql.SQLIntegrityConstraintViolationException: integrity constraint violation: foreign key no action;
FKM7JOBNABH2YROPE99IL1DO66U table: PERSON_ADDRESS
```

By simply removing the parent-side, Hibernate can safely remove the associated link records as you can see in the following example:

Example 154. Unidirectional `@ManyToMany` entity removal

```
JAVA
Person person1 = entityManager.find( Person.class, personId );
entityManager.remove( person1 );
```

```
SQL
DELETE FROM Person_Address
WHERE  Person_id = 1

DELETE FROM Person
WHERE  id = 1
```

Bidirectional `@ManyToMany`

A bidirectional `@ManyToMany` association has an owning and a `mappedBy` side. To preserve synchronicity between both sides, it's good practice to provide helper methods for adding or removing child entities.

Example 155. Bidirectional `@ManyToMany`

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();

    public Person() {
    }

    public Person(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public List<Address> getAddresses() {
        return addresses;
    }

    public void addAddress(Address address) {
        addresses.add( address );
        address.getOwners().add( this );
    }

    public void removeAddress(Address address) {
        addresses.remove( address );
        address.getOwners().remove( this );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals( registrationNumber, person.registrationNumber );
    }

    @Override
    public int hashCode() {
        return Objects.hash( registrationNumber );
    }
}

@Entity(name = "Address")
public static class Address {

    @Id
    @GeneratedValue
```

```
    private Long id;

    private String street;

    @Column(name = "`number`")
    private String number;

    private String postalCode;

    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();

    public Address() {
    }

    public Address(String street, String number, String postalCode) {
        this.street = street;
        this.number = number;
        this.postalCode = postalCode;
    }

    public Long getId() {
        return id;
    }

    public String getStreet() {
        return street;
    }

    public String getNumber() {
        return number;
    }

    public String getPostalCode() {
        return postalCode;
    }

    public List<Person> getOwners() {
        return owners;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Address address = (Address) o;
        return Objects.equals( street, address.street ) &&
                Objects.equals( number, address.number ) &&
                Objects.equals( postalCode, address.postalCode );
    }

    @Override
    public int hashCode() {
        return Objects.hash( street, number, postalCode );
```

```
    }
}
```

SQL

```
CREATE TABLE Address (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    postalCode VARCHAR(255) ,
    street VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person (
    id BIGINT NOT NULL ,
    registrationNumber VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person_Address (
    owners_id BIGINT NOT NULL ,
    addresses_id BIGINT NOT NULL
)

ALTER TABLE Person
ADD CONSTRAINT UK_23enodonj49jm8uwec4i7y37f
UNIQUE (registrationNumber)

ALTER TABLE Person_Address
ADD CONSTRAINT FKm7j0bnabh2yr0pe99il1d066u
FOREIGN KEY (addresses_id) REFERENCES Address

ALTER TABLE Person_Address
ADD CONSTRAINT FKbn86l24gmxdv2vmekayqcsgup
FOREIGN KEY (owners_id) REFERENCES Person
```

With the helper methods in place, the synchronicity management can be simplified, as you can see in the following example:

Example 156. Bidirectional `@ManyToMany` lifecycle

JAVA

```
Person person1 = new Person( "ABC-123" );
Person person2 = new Person( "DEF-456" );

Address address1 = new Address( "12th Avenue", "12A", "4005A" );
Address address2 = new Address( "18th Avenue", "18B", "4007B" );

person1.addAddress( address1 );
person1.addAddress( address2 );

person2.addAddress( address1 );

entityManager.persist( person1 );
entityManager.persist( person2 );

entityManager.flush();

person1.removeAddress( address1 );
```

SQL

```
INSERT INTO Person ( registrationNumber, id )
VALUES ( 'ABC-123', 1 )

INSERT INTO Address ( number, postalCode, street, id )
VALUES ( '12A', '4005A', '12th Avenue', 2 )

INSERT INTO Address ( number, postalCode, street, id )
VALUES ( '18B', '4007B', '18th Avenue', 3 )

INSERT INTO Person ( registrationNumber, id )
VALUES ( 'DEF-456', 4 )

INSERT INTO Person_Address ( owners_id, addresses_id )
VALUES ( 1, 2 )

INSERT INTO Person_Address ( owners_id, addresses_id )
VALUES ( 1, 3 )

INSERT INTO Person_Address ( owners_id, addresses_id )
VALUES ( 4, 2 )

DELETE FROM Person_Address
WHERE  owners_id = 1

INSERT INTO Person_Address ( owners_id, addresses_id )
VALUES ( 1, 3 )
```

If a bidirectional `@OneToMany` association performs better when removing or changing the order of child elements, the `@ManyToMany` relationship cannot benefit from such an optimization because the foreign key side is not in control. To overcome this limitation, the link table must be directly exposed and the `@ManyToMany` association split into two bidirectional `@OneToMany` relationships.

Bidirectional many-to-many with a link entity

To most natural `@ManyToMany` association follows the same logic employed by the database schema, and the link table has an associated entity which controls the relationship for both sides that need to be joined.

Example 157. Bidirectional many-to-many with link entity

JAVA

```
@Entity(name = "Person")
public static class Person implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PersonAddress> addresses = new ArrayList<>();

    public Person() {
    }

    public Person(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public Long getId() {
        return id;
    }

    public List<PersonAddress> getAddresses() {
        return addresses;
    }

    public void addAddress(Address address) {
        PersonAddress personAddress = new PersonAddress( this, address );
        addresses.add( personAddress );
        address.getOwners().add( personAddress );
    }

    public void removeAddress(Address address) {
        PersonAddress personAddress = new PersonAddress( this, address );
        address.getOwners().remove( personAddress );
        addresses.remove( personAddress );
        personAddress.setPerson( null );
        personAddress.setAddress( null );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals( registrationNumber, person.registrationNumber );
    }

    @Override
    public int hashCode() {
```

```
        return Objects.hash( registrationNumber );
    }
}

@Entity(name = "PersonAddress")
public static class PersonAddress implements Serializable {

    @Id
    @ManyToOne
    private Person person;

    @Id
    @ManyToOne
    private Address address;

    public PersonAddress() {
    }

    public PersonAddress(Person person, Address address) {
        this.person = person;
        this.address = address;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        PersonAddress that = (PersonAddress) o;
        return Objects.equals( person, that.person ) &&
                Objects.equals( address, that.address );
    }

    @Override
    public int hashCode() {
        return Objects.hash( person, address );
    }
}
```

```
@Entity(name = "Address")
public static class Address implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    @Column(name = "`number`")
    private String number;

    private String postalCode;

    @OneToMany(mappedBy = "address", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PersonAddress> owners = new ArrayList<>();

    public Address() {
    }

    public Address(String street, String number, String postalCode) {
        this.street = street;
        this.number = number;
        this.postalCode = postalCode;
    }

    public Long getId() {
        return id;
    }

    public String getStreet() {
        return street;
    }

    public String getNumber() {
        return number;
    }

    public String getPostalCode() {
        return postalCode;
    }

    public List<PersonAddress> getOwners() {
        return owners;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Address address = (Address) o;
        return Objects.equals( street, address.street ) &&
                Objects.equals( number, address.number ) &&
                Objects.equals( postalCode, address.postalCode );
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash( street, number, postalCode );
    }
}
```

SQL

```
CREATE TABLE Address (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    postalCode VARCHAR(255) ,
    street VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person (
    id BIGINT NOT NULL ,
    registrationNumber VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE PersonAddress (
    person_id BIGINT NOT NULL ,
    address_id BIGINT NOT NULL ,
    PRIMARY KEY ( person_id, address_id )
)

ALTER TABLE Person
ADD CONSTRAINT UK_23enodonj49jm8uwec4i7y37f
UNIQUE (registrationNumber)

ALTER TABLE PersonAddress
ADD CONSTRAINT FK8b3lru5fyej1aarjflamwghqq
FOREIGN KEY (person_id) REFERENCES Person

ALTER TABLE PersonAddress
ADD CONSTRAINT FK7p69mgialumhegyl4byrh65jk
FOREIGN KEY (address_id) REFERENCES Address
```

Both the `Person` and the `Address` have a `mappedBy` `@OneToMany` side, while the `PersonAddress` owns the `person` and the `address` `@ManyToOne` associations. Because this mapping is formed out of two bidirectional associations, the helper methods are even more relevant.

> The aforementioned example uses a Hibernate specific mapping for the link entity since JPA doesn't allow building a composite identifier out of multiple `@ManyToOne` associations. For more details, see the Composite identifiers - associations section.

The entity state transitions are better managed than in the previous bidirectional `@ManyToMany` case.

Example 158. Bidirectional many-to-many with link entity lifecycle

JAVA

```
Person person1 = new Person( "ABC-123" );
Person person2 = new Person( "DEF-456" );

Address address1 = new Address( "12th Avenue", "12A", "4005A" );
Address address2 = new Address( "18th Avenue", "18B", "4007B" );

entityManager.persist( person1 );
entityManager.persist( person2 );

entityManager.persist( address1 );
entityManager.persist( address2 );

person1.addAddress( address1 );
person1.addAddress( address2 );

person2.addAddress( address1 );

entityManager.flush();

log.info( "Removing address" );
person1.removeAddress( address1 );
```

```SQL
INSERT  INTO Person ( registrationNumber, id )
VALUES  ( 'ABC-123', 1 )

INSERT  INTO Person ( registrationNumber, id )
VALUES  ( 'DEF-456', 2 )

INSERT  INTO Address ( number, postalCode, street, id )
VALUES  ( '12A', '4005A', '12th Avenue', 3 )

INSERT  INTO Address ( number, postalCode, street, id )
VALUES  ( '18B', '4007B', '18th Avenue', 4 )

INSERT  INTO PersonAddress ( person_id, address_id )
VALUES  ( 1, 3 )

INSERT  INTO PersonAddress ( person_id, address_id )
VALUES  ( 1, 4 )

INSERT  INTO PersonAddress ( person_id, address_id )
VALUES  ( 2, 3 )

DELETE  FROM PersonAddress
WHERE   person_id = 1 AND address_id = 3
```

There is only one delete statement executed because, this time, the association is controlled by the `@ManyToOne` side which only has to monitor the state of the underlying foreign key relationship to trigger the right DML statement.

2.8. Collections

Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities. In this context, the distinction between value and reference semantics is very important. An object in a collection might be handled with *value* semantics (its life cycle being fully depends on the collection owner), or it might be a reference to another entity with its own life cycle. In the latter case, only the *link* between the two objects is considered to be a state held by the collection.

The owner of the collection is always an entity, even if the collection is defined by an embeddable type. Collections form one/many-to-many associations between types so there can be:

- value type collections
- embeddable type collections
- entity collections

Hibernate uses its own collection implementations which are enriched with lazy-loading, caching or state change detection semantics. For this reason, persistent collections must be declared as an interface type. The actual interface might be `java.util.Collection`, `java.util.List`, `java.util.Set`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap`

or even other object types (meaning you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`).

As the following example demonstrates, it's important to use the interface type and not the collection implementation, as declared in the entity mapping.

Example 159. Hibernate uses its own collection implementations

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    private List<String> phones = new ArrayList<>();

    public List<String> getPhones() {
        return phones;
    }
}

Person person = entityManager.find( Person.class, 1L );
//Throws java.lang.ClassCastException: org.hibernate.collection.internal.PersistentBag cannot be cast to
java.util.ArrayList
ArrayList<String> phones = (ArrayList<String>) person.getPhones();
```

> It is important that collections be defined using the appropriate Java Collections Framework interface rather than a specific implementation. From a theoretical perspective, this just follows good design principles. From a practical perspective, Hibernate (like other persistence providers) will use their own collection implementations which conform to the Java Collections Framework interfaces.

The persistent collections injected by Hibernate behave like `ArrayList`, `HashSet`, `TreeSet`, `HashMap` or `TreeMap`, depending on the interface type.

2.8.1. Collections as a value type

Value and embeddable type collections have a similar behavior as simple value types because they are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced. If a collection is passed from one persistent object to another, its elements might be moved from one table to another.

> Two entities cannot share a reference to the same collection instance. Collection-valued properties do not support null value semantics because Hibernate does not distinguish between a null collection reference and an empty collection.

2.8.2. Collections of value types

Collections of value type include basic and embeddable types. Collections cannot be nested, and, when used in collections, embeddable types are not allowed to define other collections.

For collections of value types, JPA 2.0 defines the `@ElementCollection` annotation. The lifecycle of the value-type collection is entirely controlled by its owning entity.

Considering the previous example mapping, when clearing the phone collection, Hibernate deletes all the associated phones. When adding a new element to the value type collection, Hibernate issues a new insert statement.

Example 160. Value type collection lifecycle

JAVA

```
person.getPhones().clear();
person.getPhones().add( "123-456-7890" );
person.getPhones().add( "456-000-1234" );
```

SQL

```
DELETE FROM Person_phones WHERE   Person_id = 1

INSERT INTO Person_phones ( Person_id, phones )
VALUES ( 1, '123-456-7890' )

INSERT INTO Person_phones  (Person_id, phones)
VALUES  ( 1, '456-000-1234' )
```

If removing all elements or adding new ones is rather straightforward, removing a certain entry actually requires reconstructing the whole collection from scratch.

Example 161. Removing collection elements

JAVA

```
person.getPhones().remove( 0 );
```

SQL

```
DELETE FROM Person_phones WHERE Person_id = 1

INSERT INTO Person_phones ( Person_id, phones )
VALUES ( 1, '456-000-1234' )
```

Depending on the number of elements, this behavior might not be efficient, if many elements need to be deleted and reinserted back into the database table. A workaround is to use an `@OrderColumn`, which, although not as efficient as when using the actual link table primary key, might improve the efficiency of the remove operations.

Example 162. Removing collection elements using the order column

JAVA

```
@ElementCollection
@OrderColumn(name = "order_id")
private List<String> phones = new ArrayList<>();

person.getPhones().remove( 0 );
```

SQL

```
DELETE FROM Person_phones
WHERE  Person_id = 1
       AND order_id = 1

UPDATE Person_phones
SET    phones = '456-000-1234'
WHERE  Person_id = 1
       AND order_id = 0
```

> The `@OrderColumn` column works best when removing from the tail of the collection, as it only requires a single delete statement. Removing from the head or the middle of the collection requires deleting the extra elements and updating the remaining ones to preserve element order.

Embeddable type collections behave the same way as value type collections. Adding embeddables to the collection triggers the associated insert statements and removing elements from the collection will generate delete statements.

Example 163. Embeddable type collections

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    private List<Phone> phones = new ArrayList<>();

    public List<Phone> getPhones() {
        return phones;
    }
}

@Embeddable
public static class Phone {

    private String type;

    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(String type, String number) {
        this.type = type;
        this.number = number;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }
}

person.getPhones().add( new Phone( "landline", "028-234-9876" ) );
person.getPhones().add( new Phone( "mobile", "072-122-9876" ) );
```

SQL

```
INSERT INTO Person_phones ( Person_id, number, type )
VALUES ( 1, '028-234-9876', 'landline' )

INSERT INTO Person_phones ( Person_id, number, type )
VALUES ( 1, '072-122-9876', 'mobile' )
```

2.8.3. Collections of entities

If value type collections can only form a one-to-many association between an owner entity and multiple basic or embeddable types, entity collections can represent both @OneToMany and @ManyToMany associations.

From a relational database perspective, associations are defined by the foreign key side (the child-side). With value type collections, only the entity can control the association (the parent-side), but for a collection of entities, both sides of the association are managed by the persistence context.

For this reason, entity collections can be devised into two main categories: unidirectional and bidirectional associations. Unidirectional associations are very similar to value type collections since only the parent side controls this relationship. Bidirectional associations are more tricky since, even if sides need to be in-sync at all times, only one side is responsible for managing the association. A bidirectional association has an *owning* side and an *inverse (mappedBy)* side.

Another way of categorizing entity collections is by the underlying collection type, and so we can have:

- bags
- indexed lists
- sets
- sorted sets
- maps
- sorted maps
- arrays

In the following sections, we will go through all these collection types and discuss both unidirectional and bidirectional associations.

2.8.4. Bags

Bags are unordered lists and we can have unidirectional bags or bidirectional ones.

Unidirectional bags

The unidirectional bag is mapped using a single `@OneToMany` annotation on the parent side of the association. Behind the scenes, Hibernate requires an association table to manage the parent-child relationship, as we can see in the following example:

Example 164. Unidirectional bag

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public List<Phone> getPhones() {
        return phones;
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }
}
```

```SQL
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person_Phone (
    Person_id BIGINT NOT NULL ,
    phones_id BIGINT NOT NULL
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    type VARCHAR(255) ,
    PRIMARY KEY ( id )
)

ALTER TABLE Person_Phone
ADD CONSTRAINT UK_9uhc5itwc9h5gcng944pcaslf
UNIQUE (phones_id)

ALTER TABLE Person_Phone
ADD CONSTRAINT FKr38us2n8g5p9rj0b494sd3391
FOREIGN KEY (phones_id) REFERENCES Phone

ALTER TABLE Person_Phone
ADD CONSTRAINT FK2ex4e4p7w1cj310kg2woisjl2
FOREIGN KEY (Person_id) REFERENCES Person
```

> Because both the parent and the child sides are entities, the persistence context manages each entity separately. Cascades can propagate an entity state transition from a parent entity to its children.

By marking the parent side with the `CascadeType.ALL` attribute, the unidirectional association lifecycle becomes very similar to that of a value type collection.

Example 165. Unidirectional bag lifecycle

JAVA

```
Person person = new Person( 1L );
person.getPhones().add( new Phone( 1L, "landline", "028-234-9876" ) );
person.getPhones().add( new Phone( 2L, "mobile", "072-122-9876" ) );
entityManager.persist( person );
```

SQL

```
INSERT INTO Person ( id )
VALUES ( 1 )

INSERT INTO Phone ( number, type, id )
VALUES ( '028-234-9876', 'landline', 1 )

INSERT INTO Phone ( number, type, id )
VALUES ( '072-122-9876', 'mobile', 2 )

INSERT INTO Person_Phone ( Person_id, phones_id )
VALUES ( 1, 1 )

INSERT INTO Person_Phone ( Person_id, phones_id )
VALUES ( 1, 2 )
```

In the example above, once the parent entity is persisted, the child entities are going to be persisted as well.

> Just like value type collections, unidirectional bags are not as efficient when it comes to modifying the collection structure (removing or reshuffling elements). Because the parent-side cannot uniquely identify each individual child, Hibernate might delete all child table rows associated with the parent entity and re-add them according to the current collection state.

Bidirectional bags

The bidirectional bag is the most common type of entity collection. The `@ManyToOne` side is the owning side of the bidirectional bag association, while the `@OneToMany` is the *inverse* side, being marked with the `mappedBy` attribute.

Example 166. Bidirectional bag

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public List<Phone> getPhones() {
        return phones;
    }

    public void addPhone(Phone phone) {
        phones.add( phone );
        phone.setPerson( this );
    }

    public void removePhone(Phone phone) {
        phones.remove( phone );
        phone.setPerson( null );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`", unique = true)
    @NaturalId
    private String number;

    @ManyToOne
    private Person person;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
```

```
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

SQL

```
CREATE TABLE Person (
    id BIGINT NOT NULL, PRIMARY KEY (id)
)

CREATE TABLE Phone (
    id BIGINT NOT NULL,
    number VARCHAR(255),
    type VARCHAR(255),
    person_id BIGINT,
    PRIMARY KEY (id)
)

ALTER TABLE Phone
ADD CONSTRAINT UK_l329ab0g4c1t78onljnxmbnp6
UNIQUE (number)

ALTER TABLE Phone
ADD CONSTRAINT FKmw13yfsjypiiq0i1osdkaeqpg
FOREIGN KEy (person_id) REFERENCES Person
```

Example 167. Bidirectional bag lifecycle

JAVA
```
person.addPhone( new Phone( 1L, "landline", "028-234-9876" ) );
person.addPhone( new Phone( 2L, "mobile", "072-122-9876" ) );
entityManager.flush();
person.removePhone( person.getPhones().get( 0 ) );
```

SQL
```
INSERT INTO Phone (number, person_id, type, id)
VALUES ( '028-234-9876', 1, 'landline', 1 )

INSERT INTO Phone (number, person_id, type, id)
VALUES ( '072-122-9876', 1, 'mobile', 2 )

UPDATE Phone
SET person_id = NULL, type = 'landline' where id = 1
```

Example 168. Bidirectional bag with orphan removal

JAVA
```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Phone> phones = new ArrayList<>();
```

SQL
```
DELETE FROM Phone WHERE id = 1
```

When rerunning the previous example, the child will get removed because the parent-side propagates the removal upon disassociating the child entity reference.

2.8.5. Ordered Lists

Although they use the `List` interface on the Java side, bags don't retain element order. To preserve the collection element order, there are two possibilities:

`@OrderBy`
: the collection is ordered upon retrieval using a child entity property

`@OrderColumn`
: the collection uses a dedicated order column in the collection link table

Unidirectional ordered lists

When using the `@OrderBy` annotation, the mapping looks as follows:

Example 169. Unidirectional `@OrderBy` *list*

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(cascade = CascadeType.ALL)
    @OrderBy("number")
    private List<Phone> phones = new ArrayList<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public List<Phone> getPhones() {
        return phones;
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }
}
```

The database mapping is the same as with the Unidirectional bags example, so it won't be repeated. Upon fetching the collection, Hibernate generates the following select statement:

Example 170. Unidirectional `@OrderBy` list select statement

```sql
SELECT
   phones0_.Person_id AS Person_i1_1_0_,
   phones0_.phones_id AS phones_i2_1_0_,
   unidirecti1_.id AS id1_2_1_,
   unidirecti1_."number" AS number2_2_1_,
   unidirecti1_.type AS type3_2_1_
FROM
   Person_Phone phones0_
INNER JOIN
   Phone unidirecti1_ ON phones0_.phones_id=unidirecti1_.id
WHERE
   phones0_.Person_id = 1
ORDER BY
   unidirecti1_."number"
```

The child table column is used to order the list elements.

> The `@OrderBy` annotation can take multiple entity properties, and each property can take an ordering direction too (e.g. `@OrderBy("name ASC, type DESC")`).
>
> If no property is specified (e.g. `@OrderBy`), the primary key of the child entity table is used for ordering.

Another ordering option is to use the `@OrderColumn` annotation:

Example 171. Unidirectional `@OrderColumn` list

```java
@OneToMany(cascade = CascadeType.ALL)
@OrderColumn(name = "order_id")
private List<Phone> phones = new ArrayList<>();
```

```SQL
CREATE TABLE Person_Phone (
    Person_id BIGINT NOT NULL ,
    phones_id BIGINT NOT NULL ,
    order_id INTEGER NOT NULL ,
    PRIMARY KEY ( Person_id, order_id )
)
```

This time, the link table takes the `order_id` column and uses it to materialize the collection element order. When fetching the list, the following select query is executed:

Example 172. Unidirectional `@OrderColumn` list select statement

```SQL
select
   phones0_.Person_id as Person_i1_1_0_,
   phones0_.phones_id as phones_i2_1_0_,
   phones0_.order_id as order_id3_0_,
   unidirecti1_.id as id1_2_1_,
   unidirecti1_.number as number2_2_1_,
   unidirecti1_.type as type3_2_1_
from
   Person_Phone phones0_
inner join
   Phone unidirecti1_
      on phones0_.phones_id=unidirecti1_.id
where
   phones0_.Person_id = 1
```

With the `order_id` column in place, Hibernate can order the list in-memory after it's being fetched from the database.

Bidirectional ordered lists

The mapping is similar with the Bidirectional bags example, just that the parent side is going to be annotated with either `@OrderBy` or `@OrderColumn`.

Example 173. Bidirectional `@OrderBy` list

```JAVA
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@OrderBy("number")
private List<Phone> phones = new ArrayList<>();
```

Just like with the unidirectional `@OrderBy` list, the `number` column is used to order the statement on the SQL level.

When using the `@OrderColumn` annotation, the `order_id` column is going to be embedded in the child table:

Example 174. Bidirectional `@OrderColumn` list

JAVA

```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@OrderColumn(name = "order_id")
private List<Phone> phones = new ArrayList<>();
```

SQL

```
CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    type VARCHAR(255) ,
    person_id BIGINT ,
    order_id INTEGER ,
    PRIMARY KEY ( id )
)
```

When fetching the collection, Hibernate will use the fetched ordered columns to sort the elements according to the `@OrderColumn` mapping.

2.8.6. Sets

Sets are collections that don't allow duplicate entries and Hibernate supports both the unordered `Set` and the natural-ordering `SortedSet`.

Unidirectional sets

The unidirectional set uses a link table to hold the parent-child associations and the entity mapping looks as follows:

Example 175. Unidirectional set

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(cascade = CascadeType.ALL)
    private Set<Phone> phones = new HashSet<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Set<Phone> getPhones() {
        return phones;
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
```

```
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

The unidirectional set lifecycle is similar to that of the Unidirectional bags, so it can be omitted. The only difference is that `Set` doesn't allow duplicates, but this constraint is enforced by the Java object contract rather then the database mapping.

> When using sets, it's very important to supply proper equals/hashCode implementations for child entities. In the absence of a custom equals/hashCode implementation logic, Hibernate will use the default Java reference-based object equality which might render unexpected results when mixing detached and managed object instances.

Bidirectional sets

Just like bidirectional bags, the bidirectional set doesn't use a link table, and the child table has a foreign key referencing the parent table primary key. The lifecycle is just like with bidirectional bags except for the duplicates which are filtered out.

Example 176. Bidirectional set

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private Set<Phone> phones = new HashSet<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Set<Phone> getPhones() {
        return phones;
    }

    public void addPhone(Phone phone) {
        phones.add( phone );
        phone.setPerson( this );
    }

    public void removePhone(Phone phone) {
        phones.remove( phone );
        phone.setPerson( null );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`", unique = true)
    @NaturalId
    private String number;

    @ManyToOne
    private Person person;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
```

```
        return id;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

2.8.7. Sorted sets

For sorted sets, the entity mapping must use the `SortedSet` interface instead. According to the `SortedSet` contract, all elements must implement the `Comparable` interface and therefore provide the sorting logic.

Unidirectional sorted sets

A `SortedSet` that relies on the natural sorting order given by the child element `Comparable` implementation logic must be annotated with the `@SortNatural` Hibernate annotation.

Example 177. Unidirectional natural sorted set

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(cascade = CascadeType.ALL)
    @SortNatural
    private SortedSet<Phone> phones = new TreeSet<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Set<Phone> getPhones() {
        return phones;
    }
}

@Entity(name = "Phone")
public static class Phone implements Comparable<Phone> {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }

    @Override
    public int compareTo(Phone o) {
```

```
        return number.compareTo( o.getNumber() );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

The lifecycle and the database mapping are identical to the Unidirectional bags, so they are intentionally omitted.

To provide a custom sorting logic, Hibernate also provides a `@SortComparator` annotation:

Example 178. Unidirectional custom comparator sorted set

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    @SortComparator(ReverseComparator.class)
    private SortedSet<Phone> phones = new TreeSet<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Set<Phone> getPhones() {
        return phones;
    }
}

public static class ReverseComparator implements Comparator<Phone> {
    @Override
    public int compare(Phone o1, Phone o2) {
        return o2.compareTo( o1 );
    }
}

@Entity(name = "Phone")
public static class Phone implements Comparable<Phone> {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(Long id, String type, String number) {
        this.id = id;
        this.type = type;
        this.number = number;
    }

    public Long getId() {
        return id;
    }

    public String getType() {
        return type;
```

```
    }

    public String getNumber() {
        return number;
    }

    @Override
    public int compareTo(Phone o) {
        return number.compareTo( o.getNumber() );
    }

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }

    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

Bidirectional sorted sets

The `@SortNatural` and `@SortComparator` work the same for bidirectional sorted sets too:

Example 179. Bidirectional natural sorted set

JAVA

```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@SortNatural
private SortedSet<Phone> phones = new TreeSet<>();


@OneToMany(cascade = CascadeType.ALL)
@SortComparator(ReverseComparator.class)
```

2.8.8. Maps

A `java.util.Map` is ternary association because it required a parent entity a map key and a value. An entity can either be a map key or a map value, depending on the mapping. Hibernate allows using the following map keys:

`MapKeyColumn`

for value type maps, the map key is a column in the link table that defines the grouping logic

`MapKey`

the map key is either the primary key or another property of the entity stored as a map entry value

`MapKeyEnumerated`

the map key is an `Enum` of the target child entity

`MapKeyTemporal`

the map key is a `Date` or a `Calendar` of the target child entity

`MapKeyJoinColumn`

the map key is an entity mapped as an association in the child entity that's stored as a map entry key

Value type maps

A map of value type must use the `@ElementCollection` annotation, just like value type lists, bags or sets.

Example 180. Value type map with an entity as a map key

JAVA

```
public enum PhoneType {
    LAND_LINE,
    MOBILE
}

@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    @ElementCollection
    @CollectionTable(name = "phone_register")
    @Column(name = "since")
    @MapKeyJoinColumn(name = "phone_id", referencedColumnName = "id")
    private Map<Phone, Date> phoneRegister = new HashMap<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Map<Phone, Date> getPhoneRegister() {
        return phoneRegister;
    }
}

@Embeddable
public static class Phone {

    private PhoneType type;

    @Column(name = "`number`")
    private String number;

    public Phone() {
    }

    public Phone(PhoneType type, String number) {
        this.type = type;
        this.number = number;
    }

    public PhoneType getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }
}
```

```sql
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE phone_register (
    Person_id BIGINT NOT NULL ,
    since TIMESTAMP ,
    number VARCHAR(255) NOT NULL ,
    type INTEGER NOT NULL ,
    PRIMARY KEY ( Person_id, number, type )
)

ALTER TABLE phone_register
ADD CONSTRAINT FKrmcsa34hr68of2rq8qf526mlk
FOREIGN KEY (Person_id) REFERENCES Person
```

Adding entries to the map generates the following SQL statements:

Example 181. Adding value type map entries

```java
person.getPhoneRegister().put(
    new Phone( PhoneType.LAND_LINE, "028-234-9876" ), new Date()
);
person.getPhoneRegister().put(
    new Phone( PhoneType.MOBILE, "072-122-9876" ), new Date()
);
```

```sql
INSERT INTO phone_register (Person_id, number, type, since)
VALUES (1, '072-122-9876', 1, '2015-12-15 17:16:45.311')

INSERT INTO phone_register (Person_id, number, type, since)
VALUES (1, '028-234-9876', 0, '2015-12-15 17:16:45.311')
```

Unidirectional maps

A unidirectional map exposes a parent-child association from the parent-side only. The following example shows a unidirectional map which also uses a `@MapKeyTemporal` annotation. The map key is a timestamp and it's taken from the child entity table.

Example 182. Unidirectional Map

JAVA

```
public enum PhoneType {
    LAND_LINE,
    MOBILE
}

@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinTable(
            name = "phone_register",
            joinColumns = @JoinColumn(name = "phone_id"),
            inverseJoinColumns = @JoinColumn(name = "person_id"))
    @MapKey(name = "since")
    @MapKeyTemporal(TemporalType.TIMESTAMP)
    private Map<Date, Phone> phoneRegister = new HashMap<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Map<Date, Phone> getPhoneRegister() {
        return phoneRegister;
    }

    public void addPhone(Phone phone) {
        phoneRegister.put( phone.getSince(), phone );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    private PhoneType type;

    @Column(name = "`number`")
    private String number;

    private Date since;

    public Phone() {
    }

    public Phone(PhoneType type, String number, Date since) {
        this.type = type;
        this.number = number;
        this.since = since;
```

```
    }

    public PhoneType getType() {
        return type;
    }

    public String getNumber() {
        return number;
    }

    public Date getSince() {
        return since;
    }
}
```

SQL

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    since TIMESTAMP ,
    type INTEGER ,
    PRIMARY KEY ( id )
)

CREATE TABLE phone_register (
    phone_id BIGINT NOT NULL ,
    person_id BIGINT NOT NULL ,
    PRIMARY KEY ( phone_id, person_id )
)

ALTER TABLE phone_register
ADD CONSTRAINT FKc3jajlx41lw6clbygbw8wm65w
FOREIGN KEY (person_id) REFERENCES Phone

ALTER TABLE phone_register
ADD CONSTRAINT FK6npoomh1rp660o1b55py9ndw4
FOREIGN KEY (phone_id) REFERENCES Person
```

Bidirectional maps

Like most bidirectional associations, this relationship is owned by the child-side while the parent is the inverse side and can propagate its own state transitions to the child entities. In the following example, you can see that `@MapKeyEnumerated` was used so that the `Phone` enumeration becomes the map key.

Example 183. Bidirectional Map

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    @MapKey(name = "type")
    @MapKeyEnumerated
    private Map<PhoneType, Phone> phoneRegister = new HashMap<>();

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public Map<PhoneType, Phone> getPhoneRegister() {
        return phoneRegister;
    }

    public void addPhone(Phone phone) {
        phone.setPerson( this );
        phoneRegister.put( phone.getType(), phone );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    private PhoneType type;

    @Column(name = "`number`")
    private String number;

    private Date since;

    @ManyToOne
    private Person person;

    public Phone() {
    }

    public Phone(PhoneType type, String number, Date since) {
        this.type = type;
        this.number = number;
        this.since = since;
    }

    public PhoneType getType() {
        return type;
    }
```

```
    public String getNumber() {
        return number;
    }

    public Date getSince() {
        return since;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

SQL

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    since TIMESTAMP ,
    type INTEGER ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
)

ALTER TABLE Phone
ADD CONSTRAINT FKmw13yfsjypiiq0i1osdkaeqpg
FOREIGN KEY (person_id) REFERENCES Person
```

2.8.9. Arrays

When it comes to arrays, there is quite a difference between Java arrays and relational database array types (e.g. VARRAY, ARRAY). First, not all database systems implement the SQL-99 ARRAY type, and, for this reason, Hibernate doesn't support native database array types. Second, Java arrays are relevant for basic types only since storing multiple embeddables or entities should always be done using the Java Collection API.

2.8.10. Arrays as binary

By default, Hibernate will choose a BINARY type, as supported by the current `Dialect`.

Example 184. Binary arrays

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;
    private String[] phones;

    public Person() {
    }

    public Person(Long id) {
        this.id = id;
    }

    public String[] getPhones() {
        return phones;
    }

    public void setPhones(String[] phones) {
        this.phones = phones;
    }
}
```

SQL

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    phones VARBINARY(255) ,
    PRIMARY KEY ( id )
)
```

2.8.11. Collections as basic value type

Notice how all the previous examples explicitly mark the collection attribute as either `ElementCollection`, `OneToMany` or `ManyToMany`. Collections not marked as such require a custom Hibernate `Type` and the collection elements must be stored in a single database column.

This is sometimes beneficial. Consider a use-case such as a `VARCHAR` column that represents a delimited list/set of Strings.

Example 185. Comma delimited collection

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @Type(type = "comma_delimited_strings")
    private List<String> phones = new ArrayList<>();

    public List<String> getPhones() {
        return phones;
    }
}

public class CommaDelimitedStringsJavaTypeDescriptor extends AbstractTypeDescriptor<List> {

    public static final String DELIMITER = ",";

    public CommaDelimitedStringsJavaTypeDescriptor() {
        super(
            List.class,
            new MutableMutabilityPlan<List>() {
                @Override
                protected List deepCopyNotNull(List value) {
                    return new ArrayList( value );
                }
            }
        );
    }

    @Override
    public String toString(List value) {
        return ( (List<String>) value ).stream().collect( Collectors.joining( DELIMITER ) );
    }

    @Override
    public List fromString(String string) {
        List<String> values = new ArrayList<>();
        Collections.addAll( values, string.split( DELIMITER ) );
        return values;
    }

    @Override
    public <X> X unwrap(List value, Class<X> type, WrapperOptions options) {
        return (X) toString( value );
    }

    @Override
    public <X> List wrap(X value, WrapperOptions options) {
        return fromString( (String) value );
    }
}

public class CommaDelimitedStringsType extends AbstractSingleColumnStandardBasicType<List> {

    public CommaDelimitedStringsType() {
```

```
        super(
            VarcharTypeDescriptor.INSTANCE,
            new CommaDelimitedStringsJavaTypeDescriptor()
        );
    }

    @Override
    public String getName() {
        return "comma_delimited_strings";
    }
}
```

The developer can use the comma-delimited collection like any other collection we've discussed so far and Hibernate will take care of the type transformation part. The collection itself behaves like any other basic value type, as its lifecycle is bound to its owner entity.

Example 186. Comma delimited collection lifecycle

JAVA

```
person.phones.add( "027-123-4567" );
person.phones.add( "028-234-9876" );
session.flush();
person.getPhones().remove( 0 );
```

SQL

```
INSERT INTO Person ( phones, id )
VALUES ( '027-123-4567,028-234-9876', 1 )

UPDATE Person
SET    phones = '028-234-9876'
WHERE  id = 1
```

See the Hibernate Integrations Guide for more details on developing custom value type mappings.

2.8.12. Custom collection types

If you wish to use other collection types than `List`, `Set` or `Map`, like `Queue` for instance, you have to use a custom collection type, as illustrated by the following example:

Example 187. Custom collection mapping example

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    @CollectionType( type = "org.hibernate.userguide.collections.type.QueueType")
    private Collection<Phone> phones = new LinkedList<>();

    //Getters and setters are omitted for brevity

}

@Entity(name = "Phone")
public static class Phone implements Comparable<Phone> {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity

}

public class QueueType implements UserCollectionType {

    @Override
    public PersistentCollection instantiate(
            SharedSessionContractImplementor session,
            CollectionPersister persister) throws HibernateException {
        return new PersistentQueue( session );
    }

    @Override
    public PersistentCollection wrap(
            SharedSessionContractImplementor session,
            Object collection) {
        return new PersistentQueue( session, (List) collection );
    }

    @Override
    public Iterator getElementsIterator(Object collection) {
        return ( (Queue) collection ).iterator();
    }

    @Override
    public boolean contains(Object collection, Object entity) {
        return ( (Queue) collection ).contains( entity );
    }
```

```
    @Override
    public Object indexOf(Object collection, Object entity) {
        int i = ( (List) collection ).indexOf( entity );
        return ( i < 0 ) ? null : i;
    }

    @Override
    public Object replaceElements(
            Object original,
            Object target,
            CollectionPersister persister,
            Object owner,
            Map copyCache,
            SharedSessionContractImplementor session)
            throws HibernateException {
        Queue result = (Queue) target;
        result.clear();
        result.addAll( (Queue) original );
        return result;
    }

    @Override
    public Object instantiate(int anticipatedSize) {
        return new LinkedList<>();
    }

}

public class PersistentQueue extends PersistentBag implements Queue {

    public PersistentQueue(SharedSessionContractImplementor session) {
        super( session );
    }

    public PersistentQueue(SharedSessionContractImplementor session, List list) {
        super( session, list );
    }

    @Override
    public boolean offer(Object o) {
        return add(o);
    }

    @Override
    public Object remove() {
        return poll();
    }

    @Override
    public Object poll() {
        int size = size();
        if(size > 0) {
            Object first = get(0);
            remove( 0 );
            return first;
        }
        throw new NoSuchElementException();
    }
```

```
    @Override
    public Object element() {
        return peek();
    }

    @Override
    public Object peek() {
        return size() > 0 ? get( 0 ) : null;
    }
}
```

> The reason why the `Queue` interface is not used for the entity attribute is because Hibernate only allows the following types:
>
> - `java.util.List`
> - `java.util.Set`
> - `java.util.Map`
> - `java.util.SortedSet`
> - `java.util.SortedMap`
>
> However, the custom collection type can still be customized as long as the base type is one of the aformentioned persistent types.

This way, the `Phone` collection can be used as a `java.util.Queue`:

Example 188. Custom collection example

JAVA

```
Person person = entityManager.find( Person.class, 1L );
Queue<Phone> phones = person.getPhones();
Phone head = phones.peek();
assertSame(head, phones.poll());
assertEquals( 1, phones.size() );
```

2.9. Natural Ids

Natural ids represent domain model unique identifiers that have a meaning in the real world too. Even if a natural id does not make a good primary key (surrogate keys being usually preferred), it's still useful to tell Hibernate about it. As we will see later, Hibernate provides a dedicated, efficient API for loading an entity by its natural id much like it offers for loading by its identifier (PK).

2.9.1. Natural Id Mapping

Natural ids are defined in terms of one or more persistent attributes.

Example 189. Natural id using single basic attribute

```
JAVA
@Entity
public class Company {

    @Id
    private Integer id;

    @NaturalId
    private String taxIdentifier;

    ...
}
```

Example 190. Natural id using single embedded attribute

```
JAVA
@Entity
public class PostalCarrier {

    @Id
    private Integer id;

    @NaturalId
    @Embedded
    private PostalCode postalCode;

    ...

}

@Embeddable
public class PostalCode {
    ...
}
```

Example 191. Natural id using multiple persistent attributes

JAVA

```
@Entity
public class Course {

    @Id
    private Integer id;

    @NaturalId
    @ManyToOne
    private Department department;

    @NaturalId
    private String code;

    ...
}
```

2.9.2. Natural Id API

As stated before, Hibernate provides an API for loading entities by their associate natural id. This is represented by the `org.hibernate.NaturalIdLoadAccess` contract obtained via Session#byNaturalId.

> If the entity does not define a natural id, trying to load an entity by its natural id will throw an exception.

Example 192. Using NaturalIdLoadAccess

```
Session session = ...;

Company company = session.byNaturalId( Company.class )
    .using( "taxIdentifier","abc-123-xyz" )
    .load();

PostalCarrier carrier = session.byNaturalId( PostalCarrier.class )
    .using( "postalCode",new PostalCode(... ) )
    .load();

Department department = ...;
Course course = session.byNaturalId( Course.class )
    .using( "department",department )
    .using( "code","101" )
    .load();
```
JAVA

NaturalIdLoadAccess offers 2 distinct methods for obtaining the entity:

`load()`
: obtains a reference to the entity, making sure that the entity state is initialized

`getReference()`
: obtains a reference to the entity. The state may or may not be initialized. If the entity is already associated with the current running Session, that reference (loaded or not) is returned. If the entity is not loaded in the current Session and the entity supports proxy generation, an uninitialized proxy is generated and returned, otherwise the entity is loaded from the database and returned.

`NaturalIdLoadAccess` allows loading an entity by natural id and at the same time apply a pessimistic lock. For additional details on locking, see the Locking chapter.

We will discuss the last method available on NaturalIdLoadAccess (`setSynchronizationEnabled()`) in Natural Id - Mutability and Caching.

Because the `Company` and `PostalCarrier` entities define "simple" natural ids, we can load them as follows:

Example 193. Using SimpleNaturalIdLoadAccess

```
Session session = ...;

Company company = session.bySimpleNaturalId( Company.class )
    .load( "abc-123-xyz" );

PostalCarrier carrier = session.bySimpleNaturalId( PostalCarrier.class )
    .load( new PostalCode(... ) );
```
JAVA

Here we see the use of the `org.hibernate.SimpleNaturalIdLoadAccess` contract, obtained via `Session#bySimpleNaturalId()`. `SimpleNaturalIdLoadAccess` is similar to `NaturalIdLoadAccess` except that it does not define the using method. Instead, because these "simple" natural ids are defined based on just one attribute we can directly pass the corresponding value of that natural id attribute directly to the `load()` and `getReference()` methods.

> If the entity does not define a natural id, or if the natural id is not of a "simple" type, an exception will be thrown there.

2.9.3. Natural Id - Mutability and Caching

A natural id may be mutable or immutable. By default the `@NaturalId` annotation marks an immutable natural id attribute. An immutable natural id is expected to never change its value. If the value(s) of the natural id attribute(s) change, `@NaturalId(mutable=true)` should be used instead.

Example 194. Mutable natural id

JAVA

```
@Entity
public class Person {

    @Id
    private Integer id;

    @NaturalId( mutable = true )
    private String ssn;

    ...
}
```

Within the Session, Hibernate maintains a mapping from natural id values to entity identifiers (PK) values. If natural ids values changed, it is possible for this mapping to become out of date until a flush occurs. To work around this condition, Hibernate will attempt to discover any such pending changes and adjust them when the `load()` or `getReference()` methods are executed. To be clear: this is only pertinent for mutable natural ids.

> This *discovery and adjustment* have a performance impact. If an application is certain that none of its mutable natural ids already associated with the Session have changed, it can disable that checking by calling `setSynchronizationEnabled(false)` (the default is true). This will force Hibernate to circumvent the checking of mutable natural ids.

Example 195. Mutable natural id synchronization use-case

```java
Session session=...;

Person person = session.bySimpleNaturalId( Person.class ).load( "123-45-6789" );
person.setSsn( "987-65-4321" );

...

// returns null!
person = session.bySimpleNaturalId( Person.class )
    .setSynchronizationEnabled( false )
    .load( "987-65-4321" );

// returns correctly!
person = session.bySimpleNaturalId( Person.class )
    .setSynchronizationEnabled( true )
    .load( "987-65-4321" );
```

Not only can this NaturalId-to-PK resolution be cached in the Session, but we can also have it cached in the second-level cache if second level caching is enabled.

Example 196. Natural id caching

```java
@Entity
@NaturalIdCache
public class Company {

    @Id
    private Integer id;

    @NaturalId
    private String taxIdentifier;

    ...
}
```

2.10. Dynamic Model

> JPA only acknowledges the entity model mapping so, if you are concerned about JPA provider portability, it's best to stick to the strict POJO model. On the other hand, Hibernate can work with both POJO entities as well as with dynamic entity models.

2.10.1. Dynamic mapping models

Persistent entities do not necessarily have to be represented as POJO/JavaBean classes. Hibernate also supports dynamic models (using `Map`s of `Map`s at runtime). With this approach, you do not write persistent classes, only mapping files.

A given entity has just one entity mode within a given SessionFactory. This is a change from previous versions which allowed to define multiple entity modes for an entity and to select which to load. Entity modes can now be mixed within a domain model; a dynamic entity might reference a POJO entity, and vice versa.

Example 197. Working with Dynamic Domain Models

JAVA

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

// Create a customer entity
Map<String, String>david = new HashMap<>();
david.put( "name","David" );

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

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

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

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

> The main advantage of dynamic models is quick turnaround time for prototyping without the need for entity class implementation. The main down-fall is that you lose compile-time type checking and will likely deal with many exceptions at runtime. However, as a result of the Hibernate mapping, the database schema can easily be normalized and sound, allowing to add a proper domain model implementation on top later on.
>
> It is also interesting to note that dynamic models are great for certain integration use cases as well. Envers, for example, makes extensive use of dynamic models to represent the historical data.

2.11. Inheritance

Although relational database systems don't provide support for inheritance, Hibernate provides several strategies to leverage this object-oriented trait onto domain model entities:

MappedSuperclass
: Inheritance is implemented in domain model only without reflecting it in the database schema. See MappedSuperclass.

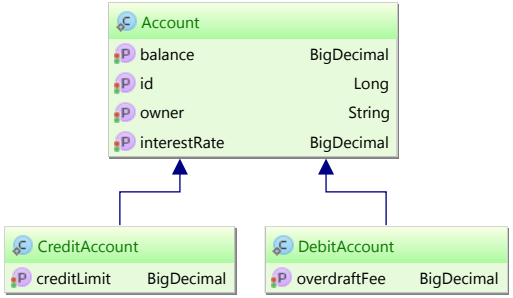
Single table
: The domain model class hierarchy is materialized into a single table which contains entities belonging to different class types. See Single table.

Joined table
: The base class and all the subclasses have their own database tables and fetching a subclass entity requires a join with the parent table as well. See Joined table.

Table per class
: Each subclass has its own table containing both the subclass and the base class properties. See Table per class.

2.11.1. MappedSuperclass

In the following domain model class hierarchy, a 'DebitAccount' and a 'CreditAccount' share the same 'Account' base class.



When using `MappedSuperclass`, the inheritance is visible in the domain model only and each database table contains both the base class and the subclass properties.

Example 198. `@MappedSuperclass` *inheritance*

JAVA

```
@MappedSuperclass
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }

    public void setOverdraftFee(BigDecimal overdraftFee) {
        this.overdraftFee = overdraftFee;
```

```
    }
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

SQL

```
CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

> Because the `@MappedSuperclass` inheritance model is not mirrored at database level, it's not possible to use polymorphic queries (fetching subclasses by their base class).

2.11.2. Single table

The single table inheritance strategy maps all subclasses to only one database table. Each subclass declares its own persistent properties. Version and id properties are assumed to be inherited from the root class.

> When omitting an explicit inheritance strategy (e.g. `@Inheritance`), JPA will choose the `SINGLE_TABLE` strategy by default.

Example 199. Single Table inheritance

JAVA

```java
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }

    public void setOverdraftFee(BigDecimal overdraftFee) {
```

```
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

SQL

```
CREATE TABLE Account (
    DTYPE VARCHAR(31) NOT NULL ,
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Each subclass in a hierarchy must define a unique discriminator value, which is used to differentiate between rows belonging to separate subclass types. If this is not specified, the `DTYPE` column is used as a discriminator, storing the associated subclass name.

Example 200. Single Table inheritance discriminator column

JAVA

```
DebitAccount debitAccount = new DebitAccount();
debitAccount.setId( 1L );
debitAccount.setOwner( "John Doe" );
debitAccount.setBalance( BigDecimal.valueOf( 100 ) );
debitAccount.setInterestRate( BigDecimal.valueOf( 1.5d ) );
debitAccount.setOverdraftFee( BigDecimal.valueOf( 25 ) );

CreditAccount creditAccount = new CreditAccount();
creditAccount.setId( 2L );
creditAccount.setOwner( "John Doe" );
creditAccount.setBalance( BigDecimal.valueOf( 1000 ) );
creditAccount.setInterestRate( BigDecimal.valueOf( 1.9d ) );
creditAccount.setCreditLimit( BigDecimal.valueOf( 5000 ) );

entityManager.persist( debitAccount );
entityManager.persist( creditAccount );
```

```sql
INSERT INTO Account (balance, interestRate, owner, overdraftFee, DTYPE, id)
VALUES (100, 1.5, 'John Doe', 25, 'DebitAccount', 1)

INSERT INTO Account (balance, interestRate, owner, creditLimit, DTYPE, id)
VALUES (1000, 1.9, 'John Doe', 5000, 'CreditAccount', 2)
```

When using polymorphic queries, only a single table is required to be scanned to fetch all associated subclass instances.

Example 201. Single Table polymorphic query

```java
List<Account> accounts = entityManager
    .createQuery( "select a from Account a" )
    .getResultList();
```

```sql
SELECT  singletabl0_.id AS id2_0_ ,
        singletabl0_.balance AS balance3_0_ ,
        singletabl0_.interestRate AS interest4_0_ ,
        singletabl0_.owner AS owner5_0_ ,
        singletabl0_.overdraftFee AS overdraf6_0_ ,
        singletabl0_.creditLimit AS creditLi7_0_ ,
        singletabl0_.DTYPE AS DTYPE1_0_
FROM    Account singletabl0_
```

> Among all other inheritance alternatives, the single table strategy performs the best since it requires access to one table only. Because all subclass columns are stored in a single table, it's not possible to use NOT NULL constraints anymore, so integrity checks must be moved either into the data access layer or enforced through `CHECK` or `TRIGGER` constraints.

Discriminator

The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the following restricted set of types as discriminator column: `String`, `char`, `int`, `byte`, `short`, `boolean` (including `yes_no`, `true_false`).

Use the `@DiscriminatorColumn` to define the discriminator column as well as the discriminator type.

The enum `DiscriminatorType` used in `javax.persistence.DiscriminatorColumn` only contains the values `STRING`, `CHAR` and `INTEGER` which means that not all Hibernate supported types are available via the `@DiscriminatorColumn` annotation. You can also use `@DiscriminatorFormula` to express in SQL a virtual discriminator column. This is particularly useful when the discriminator value can be extracted from one or more columns of the table. Both `@DiscriminatorColumn` and `@DiscriminatorFormula` are to be set on the root entity (once per persisted hierarchy).

`@org.hibernate.annotations.DiscriminatorOptions` allows to optionally specify Hibernate specific discriminator options which are not standardized in JPA. The available options are `force` and `insert`.

The `force` attribute is useful if the table contains rows with *extra* discriminator values that are not mapped to a persistent class. This could for example occur when working with a legacy database. If `force` is set to true Hibernate will specify the allowed discriminator values in the SELECT query, even when retrieving all instances of the root class.

The second option, `insert`, tells Hibernate whether or not to include the discriminator column in SQL INSERTs. Usually, the column should be part of the INSERT statement, but if your discriminator column is also part of a mapped composite identifier you have to set this option to false.

There used to be `@org.hibernate.annotations.ForceDiscriminator` annotation which was deprecated in version 3.6 and later removed. Use `@DiscriminatorOptions` instead.

Discriminator formula

Assuming a legacy database schema where the discriminator is based on inspecting a certain column, we can take advantage of the Hibernate specific `@DiscriminatorFormula` annotation and map the inheritance model as follows:

Example 202. Single Table discriminator formula

JAVA

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorFormula(
    "case when debitKey is not null " +
    "then 'Debit' " +
    "else ( " +
    "   case when creditKey is not null " +
    "   then 'Credit' " +
    "   else 'Unknown' " +
    "   end ) " +
    "end "
)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}
```

```
@Entity(name = "DebitAccount")
@DiscriminatorValue(value = "Debit")
public static class DebitAccount extends Account {

    private String debitKey;

    private BigDecimal overdraftFee;

    private DebitAccount() {
    }

    public DebitAccount(String debitKey) {
        this.debitKey = debitKey;
    }

    public String getDebitKey() {
        return debitKey;
    }

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }

    public void setOverdraftFee(BigDecimal overdraftFee) {
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
@DiscriminatorValue(value = "Credit")
public static class CreditAccount extends Account {

    private String creditKey;

    private BigDecimal creditLimit;

    private CreditAccount() {
    }

    public CreditAccount(String creditKey) {
        this.creditKey = creditKey;
    }

    public String getCreditKey() {
        return creditKey;
    }

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

```
CREATE TABLE Account (
    id int8 NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    debitKey VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    creditKey VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```
SQL

The `@DiscriminatorFormula` defines a custom SQL clause that can be used to identify a certain subclass type. The `@DiscriminatorValue` defines the mapping between the result of the `@DiscriminatorFormula` and the inheritance subclass type.

Implicit discriminator values

Aside from the usual discriminator values assigned to each individual subclass type, the `@DiscriminatorValue` can take two additional values:

`null`
: If the underlying discriminator column is null, the `null` discriminator mapping is going to be used.

`not null`
: If the underlying discriminator column has a not-null value that is not explicitly mapped to any entity, the `not-null` discriminator mapping used.

To understand how these two values work, consider the following entity mapping:

Example 203. @DiscriminatorValue `null` and `not-null` entity mapping

JAVA

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue( "null" )
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
@DiscriminatorValue( "Debit" )
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }
```

```
    public void setOverdraftFee(BigDecimal overdraftFee) {
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
@DiscriminatorValue( "Credit" )
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}

@Entity(name = "OtherAccount")
@DiscriminatorValue( "not null" )
public static class OtherAccount extends Account {

    private boolean active;

    public boolean isActive() {
        return active;
    }

    public void setActive(boolean active) {
        this.active = active;
    }
}
```

The `Account` class has a `@DiscriminatorValue( "null" )` mapping, meaning that any `account` row which does not contain any discriminator value will be mapped to an `Account` base class entity. The `DebitAccount` and `CreditAccount` entities use explicit discriminator values. The `OtherAccount` entity is used as a generic account type because it maps any database row whose discriminator column is not explicitly assigned to any other entity in the current inheritance tree.

To visualize how it works, consider the following example:

Example 204. @DiscriminatorValue `null` and `not-null` entity persistence

JAVA

```
DebitAccount debitAccount = new DebitAccount();
debitAccount.setId( 1L );
debitAccount.setOwner( "John Doe" );
debitAccount.setBalance( BigDecimal.valueOf( 100 ) );
debitAccount.setInterestRate( BigDecimal.valueOf( 1.5d ) );
debitAccount.setOverdraftFee( BigDecimal.valueOf( 25 ) );

CreditAccount creditAccount = new CreditAccount();
creditAccount.setId( 2L );
creditAccount.setOwner( "John Doe" );
creditAccount.setBalance( BigDecimal.valueOf( 1000 ) );
creditAccount.setInterestRate( BigDecimal.valueOf( 1.9d ) );
creditAccount.setCreditLimit( BigDecimal.valueOf( 5000 ) );

Account account = new Account();
account.setId( 3L );
account.setOwner( "John Doe" );
account.setBalance( BigDecimal.valueOf( 1000 ) );
account.setInterestRate( BigDecimal.valueOf( 1.9d ) );

entityManager.persist( debitAccount );
entityManager.persist( creditAccount );
entityManager.persist( account );

entityManager.unwrap( Session.class ).doWork( connection -> {
    try(Statement statement = connection.createStatement()) {
        statement.executeUpdate(
            "insert into Account (DTYPE, active, balance, interestRate, owner, id) " +
            "values ('Other', true, 25, 0.5, 'Vlad', 4)"
        );
    }
} );

Map<Long, Account> accounts = entityManager.createQuery(
    "select a from Account a", Account.class )
.getResultList()
.stream()
.collect( Collectors.toMap( Account::getId, Function.identity()));

assertEquals(4, accounts.size());
assertEquals( DebitAccount.class, accounts.get( 1L ).getClass() );
assertEquals( CreditAccount.class, accounts.get( 2L ).getClass() );
assertEquals( Account.class, accounts.get( 3L ).getClass() );
assertEquals( OtherAccount.class, accounts.get( 4L ).getClass() );
```

```
SQL

INSERT INTO Account (balance, interestRate, owner, overdraftFee, DTYPE, id)
VALUES (100, 1.5, 'John Doe', 25, 'Debit', 1)

INSERT INTO Account (balance, interestRate, owner, overdraftFee, DTYPE, id)
VALUES (1000, 1.9, 'John Doe', 5000, 'Credit', 2)

INSERT INTO Account (balance, interestRate, owner, id)
VALUES (1000, 1.9, 'John Doe', 3)

INSERT INTO Account (DTYPE, active, balance, interestRate, owner, id)
VALUES ('Other', true, 25, 0.5, 'Vlad', 4)

SELECT a.id as id2_0_,
       a.balance as balance3_0_,
       a.interestRate as interest4_0_,
       a.owner as owner5_0_,
       a.overdraftFee as overdraf6_0_,
       a.creditLimit as creditLi7_0_,
       a.active as active8_0_,
       a.DTYPE as DTYPE1_0_
FROM   Account a
```

As you can see, the `Account` entity row has a value of `NULL` in the `DTYPE` discriminator column, while the `OtherAccount` entity was saved with a `DTYPE` column value of `other` which has not explicit mapping.

2.11.3. Joined table

Each subclass can also be mapped to its own table. This is also called *table-per-subclass* mapping strategy. An inherited state is retrieved by joining with the table of the superclass.

A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier.

Example 205. Join Table

JAVA

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }

    public void setOverdraftFee(BigDecimal overdraftFee) {
```

```
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

SQL

```
CREATE TABLE Account (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    creditLimit NUMERIC(19, 2) ,
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE DebitAccount (
    overdraftFee NUMERIC(19, 2) ,
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

ALTER TABLE CreditAccount
ADD CONSTRAINT FKihw8h3j1k0w31cnyu7jcl7n7n
FOREIGN KEY (id) REFERENCES Account

ALTER TABLE DebitAccount
ADD CONSTRAINT FKia914478noepymc468kiaivqm
FOREIGN KEY (id) REFERENCES Account
```

The primary key of this table is also a foreign key to the superclass table and described by the `@PrimaryKeyJoinColumns`.

The table name still defaults to the non-qualified class name. Also, if `@PrimaryKeyJoinColumn` is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

Example 206. Join Table with `@PrimaryKeyJoinColumn`

JAVA

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }
```

```
    public void setOverdraftFee(BigDecimal overdraftFee) {
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

SQL

```
CREATE TABLE CreditAccount (
    creditLimit NUMERIC(19, 2) ,
    account_id BIGINT NOT NULL ,
    PRIMARY KEY ( account_id )
)

CREATE TABLE DebitAccount (
    overdraftFee NUMERIC(19, 2) ,
    account_id BIGINT NOT NULL ,
    PRIMARY KEY ( account_id )
)

ALTER TABLE CreditAccount
ADD CONSTRAINT FK8ulmk1wgs5x7igo37Ojt0q005
FOREIGN KEY (account_id) REFERENCES Account

ALTER TABLE DebitAccount
ADD CONSTRAINT FK7wjufa57Onoidv4omkkru06j
FOREIGN KEY (account_id) REFERENCES Account
```

When using polymorphic queries, the base class table must be joined with all subclass tables to fetch every associated subclass instance.

Example 207. Join Table polymorphic query

JAVA

```
List<Account> accounts = entityManager
    .createQuery( "select a from Account a" )
    .getResultList();
```

```sql
SELECT jointablet0_.id AS id1_0_ ,
       jointablet0_.balance AS balance2_0_ ,
       jointablet0_.interestRate AS interest3_0_ ,
       jointablet0_.owner AS owner4_0_ ,
       jointablet0_1_.overdraftFee AS overdraf1_2_ ,
       jointablet0_2_.creditLimit AS creditLi1_1_ ,
       CASE WHEN jointablet0_1_.id IS NOT NULL THEN 1
            WHEN jointablet0_2_.id IS NOT NULL THEN 2
            WHEN jointablet0_.id IS NOT NULL THEN 0
       END AS clazz_
FROM   Account jointablet0_
       LEFT OUTER JOIN DebitAccount jointablet0_1_ ON jointablet0_.id = jointablet0_1_.id
       LEFT OUTER JOIN CreditAccount jointablet0_2_ ON jointablet0_.id = jointablet0_2_.id
```

> The joined table inheritance polymorphic queries can use several JOINS which might affect performance when fetching a large number of entities.

2.11.4. Table per class

A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state.

In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

Example 208. Table per class

JAVA

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    public Long getId() {
        return id;
    }

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

    public String getOwner() {
        return owner;
    }

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

    public BigDecimal getBalance() {
        return balance;
    }

    public void setBalance(BigDecimal balance) {
        this.balance = balance;
    }

    public BigDecimal getInterestRate() {
        return interestRate;
    }

    public void setInterestRate(BigDecimal interestRate) {
        this.interestRate = interestRate;
    }
}

@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    public BigDecimal getOverdraftFee() {
        return overdraftFee;
    }

    public void setOverdraftFee(BigDecimal overdraftFee) {
```

```
        this.overdraftFee = overdraftFee;
    }
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    public BigDecimal getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(BigDecimal creditLimit) {
        this.creditLimit = creditLimit;
    }
}
```

SQL

```
CREATE TABLE Account (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)

CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

When using polymorphic queries, a UNION is required to fetch the base class table along with all subclass tables as well.

Example 209. Table per class polymorphic query

JAVA

```
List<Account> accounts = entityManager
    .createQuery( "select a from Account a" )
    .getResultList();
```

```sql
SELECT tablepercl0_.id AS id1_0_ ,
       tablepercl0_.balance AS balance2_0_ ,
       tablepercl0_.interestRate AS interest3_0_ ,
       tablepercl0_.owner AS owner4_0_ ,
       tablepercl0_.overdraftFee AS overdraf1_2_ ,
       tablepercl0_.creditLimit AS creditLi1_1_ ,
       tablepercl0_.clazz_ AS clazz_
FROM (
    SELECT    id ,
              balance ,
              interestRate ,
              owner ,
              CAST(NULL AS INT) AS overdraftFee ,
              CAST(NULL AS INT) AS creditLimit ,
              0 AS clazz_
    FROM      Account
    UNION ALL
    SELECT    id ,
              balance ,
              interestRate ,
              owner ,
              overdraftFee ,
              CAST(NULL AS INT) AS creditLimit ,
              1 AS clazz_
    FROM      DebitAccount
    UNION ALL
    SELECT    id ,
              balance ,
              interestRate ,
              owner ,
              CAST(NULL AS INT) AS overdraftFee ,
              creditLimit ,
              2 AS clazz_
    FROM      CreditAccount
) tablepercl0_
```

> Polymorphic queries require multiple UNION queries, so be aware of the performance implications of a large class hierarchy.

2.12. Immutability

Immutability can be specified for both entities and collections.

2.12.1. Entity immutability

If a specific entity is immutable, it is good practice to mark it with the `@Immutable` annotation.

Example 210. Immutable entity

```
JAVA
@Entity(name = "Event")
@Immutable
public static class Event {

    @Id
    private Long id;

    private Date createdOn;

    private String message;

    //Getters and setters are omitted for brevity

}
```

Internally, Hibernate is going to perform several optimizations, such as:

- reducing memory footprint since there is no need to retain the dehydrated state for the dirty checking mechanism
- speeding-up the Persistence Context flushing phase since immutable entities can skip the dirty checking process

Considering the following entity is persisted in the database:

Example 211. Persisting an immutable entity

```
JAVA
doInJPA( this::entityManagerFactory, entityManager -> {
    Event event = new Event();
    event.setId( 1L );
    event.setCreatedOn( new Date( ) );
    event.setMessage( "Hibernate User Guide rocks!" );

    entityManager.persist( event );
} );
```

When loading the entity and trying to change its state, Hibernate will skip any modification, therefore no SQL `UPDATE` statement is executed.

Example 212. The immutable entity ignores any update

JAVA

```
doInJPA( this::entityManagerFactory, entityManager -> {
    Event event = entityManager.find( Event.class, 1L );
    log.info( "Change event message" );
    event.setMessage( "Hibernate User Guide" );
} );
doInJPA( this::entityManagerFactory, entityManager -> {
    Event event = entityManager.find( Event.class, 1L );
    assertEquals("Hibernate User Guide rocks!", event.getMessage());
} );
```

SQL

```
SELECT e.id AS id1_0_0_,
       e.createdOn AS createdO2_0_0_,
       e.message AS message3_0_0_
FROM   event e
WHERE  e.id = 1

-- Change event message

SELECT e.id AS id1_0_0_,
       e.createdOn AS createdO2_0_0_,
       e.message AS message3_0_0_
FROM   event e
WHERE  e.id = 1
```

2.12.2. Collection immutability

Just like entities, collections can also be marked with the `@Immutable` annotation.

Considering the following entity mappings:

Example 213. Immutable collection

```java
@Entity(name = "Batch")
public static class Batch {

    @Id
    private Long id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @Immutable
    private List<Event> events = new ArrayList<>( );

    //Getters and setters are omitted for brevity

}

@Entity(name = "Event")
@Immutable
public static class Event {

    @Id
    private Long id;

    private Date createdOn;

    private String message;

    //Getters and setters are omitted for brevity

}
```

This time, not only the `Event` entity is immutable, but the `Event` collection stored by the `Batch` parent entity. Once the immutable collection is created, it can never be modified.

Example 214. Persisting an immutable collection

```
JAVA
doInJPA( this::entityManagerFactory, entityManager -> {
    Batch batch = new Batch();
    batch.setId( 1L );
    batch.setName( "Change request" );

    Event event1 = new Event();
    event1.setId( 1L );
    event1.setCreatedOn( new Date( ) );
    event1.setMessage( "Update Hibernate User Guide" );

    Event event2 = new Event();
    event2.setId( 2L );
    event2.setCreatedOn( new Date( ) );
    event2.setMessage( "Update Hibernate Getting Started Guide" );

    batch.getEvents().add( event1 );
    batch.getEvents().add( event2 );

    entityManager.persist( batch );
} );
```

The `Batch` entity is mutable. Only the `events` collection is immutable.

For instance, we can still modify the entity name:

Example 215. Changing the mutable entity

```
JAVA
doInJPA( this::entityManagerFactory, entityManager -> {
    Batch batch = entityManager.find( Batch.class, 1L );
    log.info( "Change batch name" );
    batch.setName( "Proposed change request" );
} );
```

```
SQL
SELECT b.id AS id1_0_0_,
       b.name AS name2_0_0_
FROM   Batch b
WHERE  b.id = 1

-- Change batch name

UPDATE batch
SET    name = 'Proposed change request'
WHERE  id = 1
```

However, when trying to modify the `events` collection:

Example 216. Immutable collections cannot be modified

JAVA
```
try {
    doInJPA( this::entityManagerFactory, entityManager -> {
        Batch batch = entityManager.find( Batch.class, 1L );
        batch.getEvents().clear();
    } );
}
catch ( Exception e ) {
    log.error( "Immutable collections cannot be modified" );
}
```

BASH
```
Unresolved directive in chapters/domain/immutability.adoc - include::extras/immutability/collection-immutability-update-example.log[]
```

> While immutable entity changes are simply discarded, modifying an immutable collection end up in a `HibernateException` being thrown.

3. Bootstrap

The term bootstrapping refers to initializing and starting a software component. In Hibernate, we are specifically talking about the process of building a fully functional `SessionFactory` instance or `EntityManagerFactory` instance, for JPA. The process is very different for each.

> This chapter will not focus on all the possibilities of bootstrapping. Those will be covered in each specific more-relevant chapters later on. Instead, we focus here on the API calls needed to perform the bootstrapping.

> During the bootstrap process, you might want to customize Hibernate behavior so make sure you check the Configurations section as well.

3.1. JPA Bootstrapping

Bootstrapping Hibernate as a JPA provider can be done in a JPA-spec compliant manner or using a proprietary bootstrapping approach. The standardized approach has some limitations in certain environments, but aside from those, it is **highly** recommended that you use JPA-standardized bootstrapping.

3.1.1. JPA-compliant bootstrapping

In JPA, we are ultimately interested in bootstrapping a `javax.persistence.EntityManagerFactory` instance. The JPA specification defines two primary standardized bootstrap approaches depending on how the application intends to access the `javax.persistence.EntityManager` instances from an `EntityManagerFactory`.

It uses the terms *EE* and *SE* for these two approaches, but those terms are very misleading in this context. What the JPA spec calls EE bootstrapping implies the existence of a container (EE, OSGi, etc), who'll manage and inject the persistence context on behalf of the application. What it calls SE bootstrapping is everything else. We will use the terms container-bootstrapping and application-bootstrapping in this guide.

> If you would like additional details on accessing and using `EntityManager` instances, sections 7.6 and 7.7 of the JPA 2.1 specification cover container-managed and application-managed `EntityManagers`, respectively.

For compliant container-bootstrapping, the container will build an `EntityManagerFactory` for each persistent-unit defined in the `META-INF/persistence.xml` configuration file and make that available to the application for injection via the `javax.persistence.PersistenceUnit` annotation or via JNDI lookup.

Example 217. Injecting a EntityManagerFactory

```
@PersistenceUnit
private EntityManagerFactory emf;
```

JAVA

The `META-INF/persistence.xml` file looks as follows:

Example 218. META-INF/persistence.xml configuration file

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

    <persistence-unit name="CRM">
        <description>
            Persistence unit for Hibernate User Guide
        </description>

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>org.hibernate.documentation.userguide.Document</class>

        <properties>
            <property name="javax.persistence.jdbc.driver"
                      value="org.h2.Driver" />

            <property name="javax.persistence.jdbc.url"
                      value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE" />

            <property name="javax.persistence.jdbc.user"
                      value="sa" />

            <property name="javax.persistence.jdbc.password"
                      value="" />

            <property name="hibernate.show_sql"
                      value="true" />

            <property name="hibernate.hbm2ddl.auto"
                      value="update" />
        </properties>

    </persistence-unit>

</persistence>
```

JAVA

For compliant application-bootstrapping, rather than the container building the `EntityManagerFactory` for the application, the application builds the `EntityManagerFactory` itself using the `javax.persistence.Persistence` bootstrap class. The application creates an `EntityManagerFactory` by calling the `createEntityManagerFactory` method:

Example 219. Application bootstrapped EntityManagerFactory

```
// Create an EMF for our CRM persistence-unit.
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "CRM" );
```

JAVA

> If you don't want to provide a `persistence.xml` configuration file, JPA allows you to provide all the configuration options in a PersistenceUnitInfo (http://docs.oracle.com/javaee/7/api/javax/persistence/spi/PersistenceUnitInfo.html) implementation and call HibernatePersistenceProvider.html#createContainerEntityManagerFactory (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/HibernatePersistenceProvider.html#createContainerEntityManagerFactory-javax.persistence.spi.PersistenceUnitInfo-java.util.Map-)
>
> .

3.1.2. Externalizing XML mapping files

JPA offers two mapping options:

- annotations
- XML mappings

Although annotations are much more common, there are projects were XML mappings are preferred. You can even mix annotations and XML mappings so that you can override annotation mappings with XML configurations that can be easily changed without recompiling the project source code. This is possible because if there are two conflicting mappings, the XML mappings takes precedence over its annotation counterpart.

The JPA specifications requires the XML mappings to be located on the class path:

> *An object/relational mapping XML file named `orm.xml` may be specified in the `META-INF` directory in the root of the persistence unit or in the `META-INF` directory of any jar file referenced by the `persistence.xml`.*
>
> *Alternatively, or in addition, one or more mapping files may be referenced by the mapping-file elements of the persistence-unit element. These mapping files may be present anywhere on the class path.*
>
> — *Section 8.2.1.6.2 of the JPA 2.1 Specification*

Therefore, the mapping files can reside in the application jar artifacts, or they can be stored in an external folder location with the cogitation that that location be included in the class path.

Hibernate is more lenient in this regard so you can use any external location even outside of the application configured class path.

Example 220. META-INF/persistence.xml configuration file for external XML mappings

JAVA

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

    <persistence-unit name="CRM">
        <description>
            Persistence unit for Hibernate User Guide
        </description>

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <mapping-file>file:///etc/opt/app/mappings/orm.xml</mapping-file>

        <properties>
            <property name="javax.persistence.jdbc.driver"
                      value="org.h2.Driver" />

            <property name="javax.persistence.jdbc.url"
                      value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE" />

            <property name="javax.persistence.jdbc.user"
                      value="sa" />

            <property name="javax.persistence.jdbc.password"
                      value="" />

            <property name="hibernate.show_sql"
                      value="true" />

            <property name="hibernate.hbm2ddl.auto"
                      value="update" />
        </properties>

    </persistence-unit>

</persistence>
```

In the `persistence.xml` configuration file above, the `orm.xml` XML file containing all JPA entity mappings is located in the `/etc/opt/app/mappings/` folder.

3.2. Native Bootstrapping

This section discusses the process of bootstrapping a Hibernate `SessionFactory`. Specifically it discusses the bootstrapping APIs as redesigned in 5.0. For a discussion of the legacy bootstrapping API, see Legacy Bootstrapping

3.2.1. Building the ServiceRegistry

The first step in native bootstrapping is the building of a `ServiceRegistry` holding the services Hibernate will need during bootstrapping and at run time.

Actually, we are concerned with building 2 different ServiceRegistries. First is the `org.hibernate.boot.registry.BootstrapServiceRegistry`. The `BootstrapServiceRegistry` is intended to hold services that Hibernate needs at both bootstrap and run time. This boils down to 3 services:

`org.hibernate.boot.registry.classloading.spi.ClassLoaderService`
which controls how Hibernate interacts with `ClassLoader`s

`org.hibernate.integrator.spi.IntegratorService`
which controls the management and discovery of `org.hibernate.integrator.spi.Integrator` instances.

`org.hibernate.boot.registry.selector.spi.StrategySelector`
which control how Hibernate resolves implementations of various strategy contracts. This is a very powerful service, but a full discussion of it is beyond the scope of this guide.

> If you are ok with the default behavior of Hibernate in regards to these `BootstrapServiceRegistry` services (which is quite often the case, especially in stand-alone environments), then building the `BootstrapServiceRegistry` can be skipped.

If you wish to alter how the `BootstrapServiceRegistry` is built, that is controlled through the `org.hibernate.boot.registry.BootstrapServiceRegistryBuilder`:

Example 221. Controlling `BootstrapServiceRegistry` building

```
JAVA
BootstrapServiceRegistryBuilder bootstrapRegistryBuilder =
    new BootstrapServiceRegistryBuilder();
// add a custom ClassLoader
bootstrapRegistryBuilder.applyClassLoader( customClassLoader );
// manually add an Integrator
bootstrapRegistryBuilder.applyIntegrator( customIntegrator );

BootstrapServiceRegistry bootstrapRegistry = bootstrapRegistryBuilder.build();
```

> The services of the `BootstrapServiceRegistry` cannot be extended (added to) nor overridden (replaced).

The second ServiceRegistry is the `org.hibernate.boot.registry.StandardServiceRegistry`. You will almost always need to configure the `StandardServiceRegistry`, which is done through `org.hibernate.boot.registry.StandardServiceRegistryBuilder`:

Example 222. Building a `BootstrapServiceRegistryBuilder`

```java
// An example using an implicitly built BootstrapServiceRegistry
StandardServiceRegistryBuilder standardRegistryBuilder =
    new StandardServiceRegistryBuilder();

// An example using an explicitly built BootstrapServiceRegistry
BootstrapServiceRegistry bootstrapRegistry =
    new BootstrapServiceRegistryBuilder().build();

StandardServiceRegistryBuilder standardRegistryBuilder =
    new StandardServiceRegistryBuilder( bootstrapRegistry );
```

A `StandardServiceRegistry` is also highly configurable via the StandardServiceRegistryBuilder API. See the `StandardServiceRegistryBuilder` Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/registry/StandardServiceRegistryBuilder.html) for more details.

Some specific methods of interest:

Example 223. Configuring a `MetadataSources`

```java
ServiceRegistry standardRegistry =
        new StandardServiceRegistryBuilder().build();

MetadataSources sources = new MetadataSources( standardRegistry );

// alternatively, we can build the MetadataSources without passing
// a service registry, in which case it will build a default
// BootstrapServiceRegistry to use.  But the approach shown
// above is preferred
// MetadataSources sources = new MetadataSources();

// add a class using JPA/Hibernate annotations for mapping
sources.addAnnotatedClass( MyEntity.class );

// add the name of a class using JPA/Hibernate annotations for mapping.
// differs from above in that accessing the Class is deferred which is
// important if using runtime bytecode-enhancement
sources.addAnnotatedClassName( "org.hibernate.example.Customer" );

// Read package-level metadata.
sources.addPackage( "hibernate.example" );

// Read package-level metadata.
sources.addPackage( MyEntity.class.getPackage() );

// Adds the named hbm.xml resource as a source: which performs the
// classpath lookup and parses the XML
sources.addResource( "org/hibernate/example/Order.hbm.xml" );

// Adds the named JPA orm.xml resource as a source: which performs the
// classpath lookup and parses the XML
sources.addResource( "org/hibernate/example/Product.orm.xml" );

// Read all mapping documents from a directory tree.
// Assumes that any file named *.hbm.xml is a mapping document.
sources.addDirectory( new File( ".") );

// Read mappings from a particular XML file
sources.addFile( new File( "./mapping.xml") );

// Read all mappings from a jar file.
// Assumes that any file named *.hbm.xml is a mapping document.
sources.addJar( new File( "./entities.jar") );

// Read a mapping as an application resource using the convention that a class named foo.bar.MyEntity is
// mapped by a file named foo/bar/MyEntity.hbm.xml which can be resolved as a classpath resource.
sources.addClass( MyEntity.class );
```

3.2.2. Event Listener registration

The main use cases for an `org.hibernate.integrator.spi.Integrator` right now are registering event listeners and providing services (see `org.hibernate.integrator.spi.ServiceContributingIntegrator`). With 5.0 we plan on expanding that to allow altering the metamodel describing the mapping between object and relational models.

Example 224. Configuring an event listener

```
JAVA
public class MyIntegrator implements org.hibernate.integrator.spi.Integrator {

    @Override
    public void integrate(
            Metadata metadata,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {

        // As you might expect, an EventListenerRegistry is the thing with which event
        // listeners are registered
        // It is a service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
            serviceRegistry.getService( EventListenerRegistry.class );

        // If you wish to have custom determination and handling of "duplicate" listeners,
        // you would have to add an implementation of the
        // org.hibernate.event.service.spi.DuplicationStrategy contract like this
        eventListenerRegistry.addDuplicationStrategy( new CustomDuplicationStrategy() );

        // EventListenerRegistry defines 3 ways to register listeners:

        // 1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH,
                                            DefaultAutoFlushEventListener.class );

        // 2) This form adds the specified listener(s) to the beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.PERSIST,
                                                DefaultPersistEventListener.class );

        // 3) This form adds the specified listener(s) to the end of the listener chain
        eventListenerRegistry.appendListeners( EventType.MERGE,
                                               DefaultMergeEventListener.class );
    }

    @Override
    public void disintegrate(
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {

    }
}
```

3.2.3. Building the Metadata

The second step in native bootstrapping is the building of a `org.hibernate.boot.Metadata` object containing the parsed representations of an application domain model and its mapping to a database. The first thing we obviously need to build a parsed representation is the source information to be parsed (annotated classes, `hbm.xml` files, `orm.xml` files). This is the purpose of `org.hibernate.boot.MetadataSources`:

`MetadataSources` has many other methods as well, explore its API and Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/MetadataSources.html) for more information. Also, all methods on `MetadataSources` offer fluent-style call chaining::

Example 225. Configuring a `MetadataSources` with method chaining

```
JAVA
ServiceRegistry standardRegistry =
        new StandardServiceRegistryBuilder().build();

MetadataSources sources = new MetadataSources( standardRegistry )
    .addAnnotatedClass( MyEntity.class )
    .addAnnotatedClassName( "org.hibernate.example.Customer" )
    .addResource( "org/hibernate/example/Order.hbm.xml" )
    .addResource( "org/hibernate/example/Product.orm.xml" );
```

Once we have the sources of mapping information defined, we need to build the `Metadata` object. If you are ok with the default behavior in building the Metadata then you can simply call the `buildMetadata` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/MetadataSources.html#buildMetadata--) method of the `MetadataSources`.

> Notice that a `ServiceRegistry` can be passed at a number of points in this bootstrapping process. The suggested approach is to build a `StandardServiceRegistry` yourself and pass that along to the `MetadataSources` constructor. From there, `MetadataBuilder`, `Metadata`, `SessionFactoryBuilder` and `SessionFactory` will all pick up that same `StandardServiceRegistry`.

However, if you wish to adjust the process of building `Metadata` from `MetadataSources`, you will need to use the `MetadataBuilder` as obtained via `MetadataSources#getMetadataBuilder`. `MetadataBuilder` allows a lot of control over the `Metadata` building process. See its Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/MetadataBuilder.html) for full details.

Example 226. Building Metadata via `MetadataBuilder`

```java
ServiceRegistry standardRegistry =
    new StandardServiceRegistryBuilder().build();

MetadataSources sources = new MetadataSources( standardRegistry );

MetadataBuilder metadataBuilder = sources.getMetadataBuilder();

// Use the JPA-compliant implicit naming strategy
metadataBuilder.applyImplicitNamingStrategy(
    ImplicitNamingStrategyJpaCompliantImpl.INSTANCE );

// specify the schema name to use for tables, etc when none is explicitly specified
metadataBuilder.applyImplicitSchemaName( "my_default_schema" );

// specify a custom Attribute Converter
metadataBuilder.applyAttributeConverter( myAttributeConverter );

Metadata metadata = metadataBuilder.build();
```

3.2.4. Building the SessionFactory

The final step in native bootstrapping is to build the `SessionFactory` itself. Much like discussed above, if you are ok with the default behavior of building a `SessionFactory` from a `Metadata` reference, you can simply call the `buildSessionFactory` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/Metadata.html#buildSessionFactory--) method on the `Metadata` object.

However, if you would like to adjust that building process you will need to use `SessionFactoryBuilder` as obtained via [`Metadata#getSessionFactoryBuilder` . Again, see its Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/Metadata.html#getSessionFactoryBuilder--) for more details.

Example 227. Native Bootstrapping - Putting it all together

```java
StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
    .configure( "org/hibernate/example/hibernate.cfg.xml" )
    .build();

Metadata metadata = new MetadataSources( standardRegistry )
    .addAnnotatedClass( MyEntity.class )
    .addAnnotatedClassName( "org.hibernate.example.Customer" )
    .addResource( "org/hibernate/example/Order.hbm.xml" )
    .addResource( "org/hibernate/example/Product.orm.xml" )
    .getMetadataBuilder()
    .applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )
    .build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
    .applyBeanManager( getBeanManager() )
    .build();
```

The bootstrapping API is quite flexible, but in most cases it makes the most sense to think of it as a 3 step process:

1. Build the `StandardServiceRegistry`
2. Build the `Metadata`
3. Use those 2 to build the `SessionFactory`

Example 228. Building `SessionFactory` via `SessionFactoryBuilder`

JAVA

```
StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()
        .configure( "org/hibernate/example/hibernate.cfg.xml" )
        .build();

Metadata metadata = new MetadataSources( standardRegistry )
    .addAnnotatedClass( MyEntity.class )
    .addAnnotatedClassName( "org.hibernate.example.Customer" )
    .addResource( "org/hibernate/example/Order.hbm.xml" )
    .addResource( "org/hibernate/example/Product.orm.xml" )
    .getMetadataBuilder()
    .applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )
    .build();

SessionFactoryBuilder sessionFactoryBuilder = metadata.getSessionFactoryBuilder();

// Supply an SessionFactory-level Interceptor
sessionFactoryBuilder.applyInterceptor( new CustomSessionFactoryInterceptor() );

// Add a custom observer
sessionFactoryBuilder.addSessionFactoryObservers( new CustomSessionFactoryObserver() );

// Apply a CDI BeanManager ( for JPA event listeners )
sessionFactoryBuilder.applyBeanManager( getBeanManager() );

SessionFactory sessionFactory = sessionFactoryBuilder.build();
```

4. Schema generation

Hibernate allows you to generate the database from the entity mappings.

> Although the automatic schema generation is very useful for testing and prototyping purposes, in a production environment, it's much more flexible to manage the schema using incremental migration scripts.

Traditionally, the process of generating schema from entity mapping has been called `HBM2DDL`. To get a list of Hibernate-native and JPA-specific configuration properties consider reading the Configurations section.

Considering the following Domain Model:

Example 229. Schema generation Domain Model

JAVA

```
@Entity(name = "Customer")
public class Customer {

    @Id
    private Integer id;

    private String name;

    @Basic( fetch = FetchType.LAZY )
    private UUID accountsPayableXrefId;

    @Lob
    @Basic( fetch = FetchType.LAZY )
    @LazyGroup( "lobs" )
    private Blob image;

    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public UUID getAccountsPayableXrefId() {
        return accountsPayableXrefId;
    }

    public void setAccountsPayableXrefId(UUID accountsPayableXrefId) {
        this.accountsPayableXrefId = accountsPayableXrefId;
    }

    public Blob getImage() {
        return image;
    }

    public void setImage(Blob image) {
        this.image = image;
    }
}

@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String name;
```

```
    @OneToMany(mappedBy = "author")
    private List<Book> books = new ArrayList<>(  );

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public List<Book> getBooks() {
        return books;
    }
}

@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    @NaturalId
    private String isbn;

    @ManyToOne
    private Person author;

    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

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

    public Person getAuthor() {
        return author;
    }
```

```
    public void setAuthor(Person author) {
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}
```

If the `hibernate.hbm2ddl.auto` configuration is set to `create`, Hibernate is going to generate the following database schema:

Example 230. Auto-generated database schema

```
SQL

create table Customer (
    id integer not null,
    accountsPayableXrefId binary,
    image blob,
    name varchar(255),
    primary key (id)
)

create table Book (
    id bigint not null,
    isbn varchar(255),
    title varchar(255),
    author_id bigint,
    primary key (id)
)

create table Person (
    id bigint not null,
    name varchar(255),
    primary key (id)
)

alter table Book
    add constraint UK_u31e1frmjp9mxf8k8tmp990i unique (isbn)

alter table Book
    add constraint FKrxrgiajod1le3gii8whx2doie
    foreign key (author_id)
    references Person
```

4.1. Importing script files

To customize the schema generation process, the `hibernate.hbm2ddl.import_files` configuration property must be used to provide other scripts files that Hibernate can use when the `SessionFactory` is started.

For instance, considering the following `schema-generation.sql` import file:

Example 231. Schema generation import file

```
create sequence book_sequence start with 1 increment by 1
```
JAVA

If we configure Hibernate to import the script above:

Example 232. Enabling query cache

```
<property
    name="hibernate.hbm2ddl.import_files"
    value="schema-generation.sql" />
```
XML

Hibernate is going to execute the script file after the schema is automatically generated.

4.2. Database objects

Hibernate allows you to customize the schema generation process via the HBM `database-object` element.

Considering the following HBM mapping:

Example 233. Schema generation HBM database-object

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

<hibernate-mapping>
    <database-object>
        <create>
            CREATE OR REPLACE FUNCTION sp_count_books(
                IN authorId bigint,
                OUT bookCount bigint)
                RETURNS bigint AS
            $BODY$
                BEGIN
                    SELECT COUNT(*) INTO bookCount
                    FROM book
                    WHERE author_id = authorId;
                END;
            $BODY$
            LANGUAGE plpgsql;
        </create>
        <drop></drop>
        <dialect-scope name="org.hibernate.dialect.PostgreSQL95Dialect" />
    </database-object>
</hibernate-mapping>
```

JAVA

When the `SessionFactory` is bootstrapped, Hibernate is going to execute the `database-object`, therefore creating the `sp_count_books` function.

4.3. Database-level checks

Hibernate offers the `@Check` annotation so that you can specify an arbitrary SQL CHECK constraint which can be defined as follows:

Example 234. Database check entity mapping example

```
JAVA
@Entity(name = "Book")
@Check( constraints = "CASE WHEN isbn IS NOT NULL THEN LENGTH(isbn) = 13 ELSE true END")
public static class Book {

    @Id
    private Long id;

    private String title;

    @NaturalId
    private String isbn;

    private Double price;

    //Getters and setters omitted for brevity

}
```

Now, if you try to add a `Book` entity with an `isbn` attribute whose length is not 13 characters, a `ConstraintViolationException` is going to be thrown.

Example 235. Database check failure example

```
JAVA
Book book = new Book();
book.setId( 1L );
book.setPrice( 49.99d );
book.setTitle( "High-Performance Java Persistence" );
book.setIsbn( "11-11-2016" );

entityManager.persist( book );
```

```
SQL
INSERT  INTO Book (isbn, price, title, id)
VALUES  ('11-11-2016', 49.99, 'High-Performance Java Persistence', 1)

-- WARN SqlExceptionHelper:129 - SQL Error: 0, SQLState: 23514
-- ERROR SqlExceptionHelper:131 - ERROR: new row for relation "book" violates check constraint "book_isbn_check"
```

4.4. Default value for database column

With Hibernate, you can specify a default value for a given database column using the `@ColumnDefault` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ColumnDefault.html) annotation.

Example 236. `@ColumnDefault` mapping example

JAVA

```
@Entity(name = "Person")
@DynamicInsert
public static class Person {

    @Id
    private Long id;

    @ColumnDefault("'N/A'")
    private String name;

    @ColumnDefault("-1")
    private Long clientId;

    //Getter and setters omitted for brevity

}
```

SQL

```
CREATE TABLE Person (
  id BIGINT NOT NULL,
  clientId BIGINT DEFAULT -1,
  name VARCHAR(255) DEFAULT 'N/A',
  PRIMARY KEY (id)
)
```

In the mapping above, both the `name` and `clientId` table columns are going to use a `DEFAULT` value.

> The entity is annotated with the `@DynamicInsert` annotation so that the `INSERT` statement does not include the entity attribute that have not been set.

This way, when omitting the `name` and the `clientId` attribute, the database is going to set them according to their default values.

Example 237. `@ColumnDefault` *mapping example*

```
doInJPA( this::entityManagerFactory, entityManager -> {
    Person person = new Person();
    person.setId( 1L );
    entityManager.persist( person );
} );
doInJPA( this::entityManagerFactory, entityManager -> {
    Person person = entityManager.find( Person.class, 1L );
    assertEquals( "N/A", person.getName() );
    assertEquals( Long.valueOf( -1L ), person.getClientId() );
} );
```

JAVA

```
INSERT INTO Person (id) VALUES (?)
```

SQL

5. Persistence Contexts

Both the `org.hibernate.Session` API and `javax.persistence.EntityManager` API represent a context for dealing with persistent data. This concept is called a `persistence context`. Persistent data has a state in relation to both a persistence context and the underlying database.

`transient`

the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and typically no identifier value has been assigned (unless the *assigned* generator was used).

`managed, or persistent`

the entity has an associated identifier and is associated with a persistence context. It may or may not physically exist in the database yet.

`detached`

the entity has an associated identifier, but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context)

`removed`

the entity has an associated identifier and is associated with a persistence context, however it is scheduled for removal from the database.

Much of the `org.hibernate.Session` and `javax.persistence.EntityManager` methods deal with moving entities between these states.

5.1. Accessing Hibernate APIs from JPA

JPA defines an incredibly useful method to allow applications access to the APIs of the underlying provider.

Example 238. Accessing Hibernate APIs from JPA

```
JAVA
Session session = entityManager.unwrap( Session.class );
SessionImplementor sessionImplementor = entityManager.unwrap( SessionImplementor.class );

SessionFactory sessionFactory = entityManager.getEntityManagerFactory().unwrap( SessionFactory.class );
```

5.2. Bytecode Enhancement

Hibernate "grew up" not supporting bytecode enhancement at all. At that time, Hibernate only supported proxy-based for lazy loading and always used diff-based dirty calculation. Hibernate 3.x saw the first attempts at bytecode enhancement support in Hibernate. We consider those initial attempts (up until 5.0) completely as an incubation. The support for bytecode enhancement in 5.0 onward is what we are discussing here.

5.2.1. Capabilities

Hibernate supports the enhancement of an application Java domain model for the purpose of adding various persistence-related capabilities directly into the class.

Lazy attribute loading

Think of this as partial loading support. Essentially you can tell Hibernate that only part(s) of an entity should be loaded upon fetching from the database and when the other part(s) should be loaded as well. Note that this is very much different from proxy-based idea of lazy loading which is entity-centric where the entity's state is loaded at once as needed. With bytecode enhancement, individual attributes or groups of attributes are loaded as needed.

Lazy attributes can be designated to be loaded together and this is called a "lazy group". By default, all singular attributes are part of a single group, meaning that when one lazy singular attribute is accessed all lazy singular attributes are loaded. Lazy plural attributes, by default, are each a lazy group by themselves. This behavior is explicitly controllable through the `@org.hibernate.annotations.LazyGroup` annotation.

Example 239. `@LazyGroup` example

JAVA

```
@Entity
public class Customer {

    @Id
    private Integer id;

    private String name;

    @Basic( fetch = FetchType.LAZY )
    private UUID accountsPayableXrefId;

    @Lob
    @Basic( fetch = FetchType.LAZY )
    @LazyGroup( "lobs" )
    private Blob image;

    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public UUID getAccountsPayableXrefId() {
        return accountsPayableXrefId;
    }

    public void setAccountsPayableXrefId(UUID accountsPayableXrefId) {
        this.accountsPayableXrefId = accountsPayableXrefId;
    }

    public Blob getImage() {
        return image;
    }

    public void setImage(Blob image) {
        this.image = image;
    }
}
```

In the above example we have 2 lazy attributes: `accountsPayableXrefId` and `image`. Each is part of a different fetch group (accountsPayableXrefId is part of the default fetch group), which means that accessing `accountsPayableXrefId` will not force the loading of image, and vice-versa.

> As a hopefully temporary legacy hold-over, it is currently required that all lazy singular associations (many-to-one and one-to-one) also include `@LazyToOne(LazyToOneOption.NO_PROXY)`. The plan is to relax that requirement later.

In-line dirty tracking

Historically Hibernate only supported diff-based dirty calculation for determining which entities in a persistence context have changed. This essentially means that Hibernate would keep track of the last known state of an entity in regards to the database (typically the last read or write). Then, as part of flushing the persistence context, Hibernate would walk every entity associated with the persistence context and check its current state against that "last known database state". This is by far the most thorough approach to dirty checking because it accounts for data-types that can change their internal state (`java.util.Date` is the prime example of this). However, in a persistence context with a large number of associated entities it can also be a performance-inhibiting approach.

If your application does not need to care about "internal state changing data-type" use cases, bytecode-enhanced dirty tracking might be a worthwhile alternative to consider, especially in terms of performance. In this approach Hibernate will manipulate the bytecode of your classes to add "dirty tracking" directly to the entity, allowing the entity itself to keep track of which of its attributes have changed. During flush time, Hibernate simply asks your entity what has changed rather that having to perform the state-diff calculations.

Bidirectional association management

Hibernate strives to keep your application as close to "normal Java usage" (idiomatic Java) as possible. Consider a domain model with a normal `Person` / `Book` bidirectional association:

Example 240. Bidirectional association

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books = new ArrayList<>(  );

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public List<Book> getBooks() {
        return books;
    }
}

@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    @NaturalId
    private String isbn;

    @ManyToOne
    private Person author;

    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
```

```
    }

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

    public Person getAuthor() {
        return author;
    }

    public void setAuthor(Person author) {
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}
```

Example 241. Incorrect normal Java usage

JAVA

```
Person person = new Person();
person.setName( "John Doe" );

Book book = new Book();
person.getBooks().add( book );
try {
    book.getAuthor().getName();
}
catch (NullPointerException expected) {
    // This blows up ( NPE ) in normal Java usage
}
```

This blows up in normal Java usage. The correct normal Java usage is:

Example 242. Correct normal Java usage

```java
Person person = new Person();
person.setName( "John Doe" );

Book book = new Book();
person.getBooks().add( book );
book.setAuthor( person );

book.getAuthor().getName();
```

Bytecode-enhanced bi-directional association management makes that first example work by managing the "other side" of a bi-directional association whenever one side is manipulated.

Internal performance optimizations

Additionally, we use the enhancement process to add some additional code that allows us to optimized certain performance characteristics of the persistence context. These are hard to discuss without diving into a discussion of Hibernate internals.

5.2.2. Performing enhancement

Run-time enhancement

Currently, run-time enhancement of the domain model is only supported in managed JPA environments following the JPA-defined SPI for performing class transformations. Even then, this support is disabled by default. To enable run-time enhancement, specify `hibernate.ejb.use_class_enhancer = true` as a persistent unit property.

> Also, at the moment, only annotated classes are supported for run-time enhancement.

Gradle plugin

Hibernate provides a Gradle plugin that is capable of providing build-time enhancement of the domain model as they are compiled as part of a Gradle build. To use the plugin a project would first need to apply it:

Example 243. Apply the Gradle plugin

```
GRADLE

ext {
    hibernateVersion = 'hibernate-version-you-want'
}

buildscript {
    dependencies {
        classpath "org.hibernate:hibernate-gradle-plugin:$hibernateVersion"
    }
}

hibernate {
    enhance {
        // any configuration goes here
    }
}
```

The configuration that is available is exposed through a registered Gradle DSL extension:

enableLazyInitialization

Whether enhancement for lazy attribute loading should be done.

enableDirtyTracking

Whether enhancement for self-dirty tracking should be done.

enableAssociationManagement

Whether enhancement for bi-directional association management should be done.

The default value for all 3 configuration settings is `false`

The `enhance { }` block is required in order for enhancement to occur. Enhancement is disabled by default in preparation for additions capabilities (hbm2ddl, etc) in the plugin.

Maven plugin

Hibernate provides a Maven plugin capable of providing build-time enhancement of the domain model as they are compiled as part of a Maven build. See the section on the Gradle plugin for details on the configuration settings. Again, the default for those 3 is `false`.

The Maven plugin supports one additional configuration settings: failOnError, which controls what happens in case of error. Default behavior is to fail the build, but it can be set so that only a warning is issued.

Example 244. Apply the Maven plugin

```xml
<build>
    <plugins>
        [...]
        <plugin>
            <groupId>org.hibernate.orm.tooling</groupId>
            <artifactId>hibernate-enhance-maven-plugin</artifactId>
            <version>$currentHibernateVersion</version>
            <executions>
                <execution>
                    <configuration>
                        <failOnError>true</failOnError>
                        <enableLazyInitialization>true</enableLazyInitialization>
                        <enableDirtyTracking>true</enableDirtyTracking>
                        <enableAssociationManagement>true</enableAssociationManagement>
                    </configuration>
                    <goals>
                        <goal>enhance</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        [...]
    </plugins>
</build>
```

5.3. Making entities persistent

Once you've created a new entity instance (using the standard `new` operator) it is in `new` state. You can make it persistent by associating it to either a `org.hibernate.Session` or `javax.persistence.EntityManager`.

Example 245. Making an entity persistent with JPA

```java
Person person = new Person();
person.setId( 1L );
person.setName("John Doe");

entityManager.persist( person );
```

Example 246. Making an entity persistent with Hibernate API

```java
Person person = new Person();
person.setId( 1L );
person.setName("John Doe");

session.save( person );
```

`org.hibernate.Session` also has a method named persist which follows the exact semantic defined in the JPA specification for the persist method. It is this `org.hibernate.Session` method to which the Hibernate `javax.persistence.EntityManager` implementation delegates.

If the `DomesticCat` entity type has a generated identifier, the value is associated with the instance when the save or persist is called. If the identifier is not automatically generated, the manually assigned (usually natural) key value has to be set on the instance before the save or persist methods are called.

5.4. Deleting (removing) entities

Entities can also be deleted.

Example 247. Deleting an entity with JPA

```
entityManager.remove( person );
```
JAVA

Example 248. Deleting an entity with Hibernate API

```
session.delete( person );
```
JAVA

> Hibernate itself can handle deleting detached state. JPA, however, disallows it. The implication here is that the entity instance passed to the `org.hibernate.Session` delete method can be either in managed or detached state, while the entity instance passed to remove on `javax.persistence.EntityManager` must be in managed state.

5.5. Obtain an entity reference without initializing its data

Sometimes referred to as lazy loading, the ability to obtain a reference to an entity without having to load its data is hugely important. The most common case being the need to create an association between an entity and another existing entity.

Example 249. Obtaining an entity reference without initializing its data with JPA

```
Book book = new Book();
book.setAuthor( entityManager.getReference( Person.class, personId ) );
```
JAVA

Example 250. Obtaining an entity reference without initializing its data with Hibernate API

```
Book book = new Book();
book.setId( 1L );
book.setIsbn( "123-456-7890" );
entityManager.persist( book );
book.setAuthor( session.load( Person.class, personId ) );
```
JAVA

The above works on the assumption that the entity is defined to allow lazy loading, generally through use of runtime proxies. In both cases an exception will be thrown later if the given entity does not refer to actual database state when the application attempts to use the returned proxy in any way that requires access to its data.

5.6. Obtain an entity with its data initialized

It is also quite common to want to obtain an entity along with its data (e.g. like when we need to display it in the UI).

Example 251. Obtaining an entity reference with its data initialized with JPA

```
Person person = entityManager.find( Person.class, personId );
```
JAVA

Example 252. Obtaining an entity reference with its data initialized with Hibernate API

```
Person person = session.get( Person.class, personId );
```
JAVA

Example 253. Obtaining an entity reference with its data initialized using the `byId()` Hibernate API

```
Person person = session.byId( Person.class ).load( personId );
```
JAVA

In both cases null is returned if no matching database row was found.

It's possible to return a Java 8 `Optional` as well:

Example 254. Obtaining an Optional entity reference with its data initialized using the `byId()` Hibernate API

```
Optional<Person> optionalPerson = session.byId( Person.class ).loadOptional( personId );
```
JAVA

5.7. Obtain an entity by natural-id

In addition to allowing to load by identifier, Hibernate allows applications to load by declared natural identifier.

Example 255. Natural-id mapping

JAVA

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    @NaturalId
    private String isbn;

    @ManyToOne
    private Person author;

    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

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

    public Person getAuthor() {
        return author;
    }

    public void setAuthor(Person author) {
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}
```

We can also opt to fetch the entity or just retrieve a reference to it when using the natural identifier loading methods.

Example 256. Get entity reference by simple natural-id

```
Book book = session.bySimpleNaturalId( Book.class ).getReference( isbn );
```
JAVA

Example 257. Load entity by natural-id

```
Book book = session
    .byNaturalId( Book.class )
    .using( "isbn", isbn )
    .load( );
```
JAVA

We can also use a Java 8 `Optional` to load an entity by its natural id:

Example 258. Load an Optional entity by natural-id

```
Optional<Book> optionalBook = session
    .byNaturalId( Book.class )
    .using( "isbn", isbn )
    .loadOptional( );
```
JAVA

Hibernate offer a consistent API for accessing persistent data by identifier or by the natural-id. Each of these defines the same two data access methods:

getReference

Should be used in cases where the identifier is assumed to exist, where non-existence would be an actual error. Should never be used to test existence. That is because this method will prefer to create and return a proxy if the data is not already associated with the Session rather than hit the database. The quintessential use-case for using this method is to create foreign-key based associations.

load

Will return the persistent data associated with the given identifier value or null if that identifier does not exist.

Each of these two methods define an overloading variant accepting a `org.hibernate.LockOptions` argument. Locking is discussed in a separate chapter.

5.8. Modifying managed/persistent state

Entities in managed/persistent state may be manipulated by the application and any changes will be automatically detected and persisted when the persistence context is flushed. There is no need to call a particular method to make your modifications persistent.

Example 259. Modifying managed state with JPA

```
Person person = entityManager.find( Person.class, personId );
person.setName("John Doe");
entityManager.flush();
```

JAVA

Example 260. Modifying managed state with Hibernate API

```
Person person = session.byId( Person.class ).load( personId );
person.setName("John Doe");
entityManager.flush();
```

JAVA

5.9. Refresh entity state

You can reload an entity instance and its collections at any time.

Example 261. Refreshing entity state with JPA

```
Person person = entityManager.find( Person.class, personId );

entityManager.createQuery( "update Person set name = UPPER(name)" ).executeUpdate();

entityManager.refresh( person );
assertEquals("JOHN DOE", person.getName() );
```

JAVA

Example 262. Refreshing entity state with Hibernate API

```java
Person person = session.byId( Person.class ).load( personId );

session.doWork( connection -> {
    try(Statement statement = connection.createStatement()) {
        statement.executeUpdate( "UPDATE Person SET name = UPPER(name)" );
    }
} );

session.refresh( person );
assertEquals("JOHN DOE", person.getName() );
```

One case where this is useful is when it is known that the database state has changed since the data was read. Refreshing allows the current database state to be pulled into the entity instance and the persistence context.

Another case where this might be useful is when database triggers are used to initialize some of the properties of the entity.

> Only the entity instance and its value type collections are refreshed unless you specify `REFRESH` as a cascade style of any associations. However, please note that Hibernate has the capability to handle this automatically through its notion of generated properties. See the discussion of non-identifier generated attributes.

> Traditionally, Hibernate has been allowing detached entities to be refreshed. Unfortunately, JPA prohibits this practice and specifies that an `IllegalArgumentException` should be thrown instead.
>
> For this reason, when bootstrapping the Hibernate `SessionFactory` using the native API, the legacy detached entity refresh behavior is going to be preserved. On the other hand, when bootstrapping Hibernate through JPA `EntityManagerFactory` building process, detached entities are not allowed to be refreshed by default.
>
> However, this default behavior can be overwritten through the `hibernate.allow_refresh_detached_entity` configuration property. If this property is explicitly set to `true`, then you can refresh detached entities even when using the JPA bootstraps mechanism, therefore bypassing the JPA specification restriction.
>
> For more about the `hibernate.allow_refresh_detached_entity` configuration property, check out the Configurations section as well.

5.9.1. Refresh gotchas

The `refresh` entity state transition is meant to overwrite the entity attributes according to the info currently contained in the associated database record.

However, you have to be very careful when cascading the refresh action to any transient entity.

For instance, consider the following example:

Example 263. Refreshing entity state gotcha

JAVA

```
try {
    Person person = entityManager.find( Person.class, personId );

    Book book = new Book();
    book.setId( 100L );
    book.setTitle( "Hibernate User Guide" );
    book.setAuthor( person );
    person.getBooks().add( book );

    entityManager.refresh( person );
}
catch ( EntityNotFoundException expected ) {
    log.info( "Beware when cascading the refresh associations to transient entities!" );
}
```

In the aforementioned example, an `EntityNotFoundException` is thrown because the `Book` entity is still in a transient state. When the refresh action is cascaded from the `Person` entity, Hibernate will not be able to locate the `Book` entity in the database.

For this reason, you should be very careful when mixing the refresh action with transient child entity objects.

5.10. Working with detached data

Detachment is the process of working with data outside the scope of any persistence context. Data becomes detached in a number of ways. Once the persistence context is closed, all data that was associated with it becomes detached. Clearing the persistence context has the same effect. Evicting a particular entity from the persistence context makes it detached. And finally, serialization will make the deserialized form be detached (the original instance is still managed).

Detached data can still be manipulated, however the persistence context will no longer automatically know about these modification and the application will need to intervene to make the changes persistent again.

5.10.1. Reattaching detached data

Reattachment is the process of taking an incoming entity instance that is in detached state and re-associating it with the current persistence context.

> JPA does not provide for this model. This is only available through Hibernate `org.hibernate.Session`.

Example 264. Reattaching a detached entity using `lock`

```
Person person = session.byId( Person.class ).load( personId );
//Clear the Session so the person entity becomes detached
session.clear();
person.setName( "Mr. John Doe" );

session.lock( person, LockMode.NONE );
```

JAVA

Example 265. Reattaching a detached entity using `saveOrUpdate`

```
Person person = session.byId( Person.class ).load( personId );
//Clear the Session so the person entity becomes detached
session.clear();
person.setName( "Mr. John Doe" );

session.saveOrUpdate( person );
```

JAVA

> The method name `update` is a bit misleading here. It does not mean that an `SQL UPDATE` is immediately performed. It does, however, mean that an `SQL UPDATE` will be performed when the persistence context is flushed since Hibernate does not know its previous state against which to compare for changes. If the entity is mapped with `select-before-update`, Hibernate will pull the current state from the database and see if an update is needed.

Provided the entity is detached, `update` and `saveOrUpdate` operate exactly the same.

5.10.2. Merging detached data

Merging is the process of taking an incoming entity instance that is in detached state and copying its data over onto a new managed instance.

Although not exactly per se, the following example is a good visualization of the `merge` operation internals.

Example 266. Visualizing merge

```
public Person merge(Person detached) {
    Person newReference = session.byId( Person.class ).load( detached.getId() );
    newReference.setName( detached.getName() );
    return newReference;
}
```

JAVA

Example 267. Merging a detached entity with JPA

```java
Person person = entityManager.find( Person.class, personId );
//Clear the EntityManager so the person entity becomes detached
entityManager.clear();
person.setName( "Mr. John Doe" );

person = entityManager.merge( person );
```

Example 268. Merging a detached entity with Hibernate API

```java
Person person = session.byId( Person.class ).load( personId );
//Clear the Session so the person entity becomes detached
session.clear();
person.setName( "Mr. John Doe" );

person = (Person) session.merge( person );
```

Merging gotchas

For example, Hibernate throws `IllegalStateException` when merging a parent entity which has references to 2 detached child entities `child1` and `child2` (obtained from different sessions), and `child1` and `child2` represent the same persistent entity, `Child`.

A new configuration property, `hibernate.event.merge.entity_copy_observer`, controls how Hibernate will respond when multiple representations of the same persistent entity ("entity copy") is detected while merging.

The possible values are:

disallow (the default)
: throws `IllegalStateException` if an entity copy is detected

allow
: performs the merge operation on each entity copy that is detected

log
: (provided for testing only) performs the merge operation on each entity copy that is detected and logs information about the entity copies. This setting requires DEBUG logging be enabled for `org.hibernate.event.internal.EntityCopyAllowedLoggedObserver`.

In addition, the application may customize the behavior by providing an implementation of `org.hibernate.event.spi.EntityCopyObserver` and setting `hibernate.event.merge.entity_copy_observer` to the class name. When this property is set to `allow` or `log`, Hibernate will merge each entity copy detected while cascading the merge operation. In the process of merging each entity copy, Hibernate will cascade the merge operation from each entity copy to its associations with `cascade=CascadeType.MERGE` or `CascadeType.ALL`. The entity state resulting from merging an entity copy will be overwritten when another entity copy is merged.

> Because cascade order is undefined, the order in which the entity copies are merged is undefined. As a result, if property values in the entity copies are not consistent, the resulting entity state will be indeterminate, and data will be lost from all entity copies except for the last one merged. Therefore, the **last writer wins**.
>
> If an entity copy cascades the merge operation to an association that is (or contains) a new entity, that new entity will be merged (i.e., persisted and the merge operation will be cascaded to its associations according to its mapping), even if that same association is ultimately overwritten when Hibernate merges a different representation having a different value for its association.
>
> If the association is mapped with `orphanRemoval = true`, the new entity will not be deleted because the semantics of orphanRemoval do not apply if the entity being orphaned is a new entity.
>
> There are known issues when representations of the same persistent entity have different values for a collection. See HHH-9239 (https://hibernate.atlassian.net/browse/HHH-9239) and HHH-9240 (https://hibernate.atlassian.net/browse/HHH-9240) for more details. These issues can cause data loss or corruption.
>
> By setting `hibernate.event.merge.entity_copy_observer` configuration property to `allow` or `log`, Hibernate will allow entity copies of any type of entity to be merged.
>
> The only way to exclude particular entity classes or associations that contain critical data is to provide a custom implementation of `org.hibernate.event.spi.EntityCopyObserver` with the desired behavior, and setting `hibernate.event.merge.entity_copy_observer` to the class name.

Hibernate provides limited DEBUG logging capabilities that can help determine the entity classes for which entity copies were found. By setting `hibernate.event.merge.entity_copy_observer` to `log` and enabling DEBUG logging for `org.hibernate.event.internal.EntityCopyAllowedLoggedObserver`, the following will be logged each time an application calls `EntityManager.merge( entity )` or `Session.merge( entity )`:

- number of times multiple representations of the same persistent entity was detected summarized by entity name;
- details by entity name and ID, including output from calling toString() on each representation being merged as well as the merge result.

The log should be reviewed to determine if multiple representations of entities containing critical data are detected. If so, the application should be modified so there is only one representation, and a custom implementation of `org.hibernate.event.spi.EntityCopyObserver` should be provided to disallow entity copies for entities with critical data.

Using optimistic locking is recommended to detect if different representations are from different versions of the same persistent entity. If they are not from the same version, Hibernate will throw either the JPA `OptimisticLockException` or the native `StaleObjectStateException` depending on your bootstrapping strategy.

5.11. Checking persistent state

An application can verify the state of entities and collections in relation to the persistence context.

Example 269. Verifying managed state with JPA

```
boolean contained = entityManager.contains( person );
```
JAVA

Example 270. Verifying managed state with Hibernate API

```
boolean contained = session.contains( person );
```
JAVA

Example 271. Verifying laziness with JPA

```
PersistenceUnitUtil persistenceUnitUtil = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();

boolean personInitialized = persistenceUnitUtil.isLoaded( person );

boolean personBooksInitialized = persistenceUnitUtil.isLoaded( person.getBooks() );

boolean personNameInitialized = persistenceUnitUtil.isLoaded( person, "name" );
```
JAVA

Example 272. Verifying laziness with Hibernate API

```
boolean personInitialized = Hibernate.isInitialized( person );

boolean personBooksInitialized = Hibernate.isInitialized( person.getBooks() );

boolean personNameInitialized = Hibernate.isPropertyInitialized( person, "name" );
```
JAVA

In JPA there is an alternative means to check laziness using the following `javax.persistence.PersistenceUtil` pattern (which is recommended wherever possible).

Example 273. Alternative JPA means to verify laziness

```
PersistenceUtil persistenceUnitUtil = Persistence.getPersistenceUtil();

boolean personInitialized = persistenceUnitUtil.isLoaded( person );

boolean personBooksInitialized = persistenceUnitUtil.isLoaded( person.getBooks() );

boolean personNameInitialized = persistenceUnitUtil.isLoaded( person, "name" );
```
JAVA

5.12. Evicting entities

When the `flush()` method is called, the state of the entity is synchronized with the database. If you do not want this synchronization to occur, or if you are processing a huge number of objects and need to manage memory efficiently, the `evict()` method can be used to remove the object and its collections from the first-level cache.

Example 274. Detaching an entity from the `EntityManager`

```java
for(Person person : entityManager.createQuery("select p from Person p", Person.class)
        .getResultList()) {
    dtos.add(toDTO(person));
    entityManager.detach( person );
}
```

Example 275. Evicting an entity from the Hibernate `Session`

```java
Session session = entityManager.unwrap( Session.class );
for(Person person : (List<Person>) session.createQuery("select p from Person p").list()) {
    dtos.add(toDTO(person));
    session.evict( person );
}
```

To detach all entities from the current persistence context, both the `EntityManager` and the Hibernate `Session` define a `clear()` method.

Example 276. Clearing the persistence context

```java
entityManager.clear();

session.clear();
```

To verify if an entity instance is currently attached to the running persistence context, both the `EntityManager` and the Hibernate `Session` define a `contains(Object entity)` method.

Example 277. Verify if an entity is contained in a persistence context

```java
entityManager.contains( person );

session.contains( person );
```

5.13. Cascading entity state transitions

JPA allows you to propagate the state transition from a parent entity to a child. For this purpose, the JPA `javax.persistence.CascadeType` defines various cascade types:

`ALL`
cascades all entity state transitions

`PERSIST`
cascades the entity persist operation.

`MERGE`
cascades the entity merge operation.

`REMOVE`
cascades the entity remove operation.

`REFRESH`
cascades the entity refresh operation.

`DETACH`
cascades the entity detach operation.

Additionally, the `CascadeType.ALL` will propagate any Hibernate-specific operation, which is defined by the `org.hibernate.annotations.CascadeType` enum:

`SAVE_UPDATE`
cascades the entity saveOrUpdate operation.

`REPLICATE`
cascades the entity replicate operation.

`LOCK`
cascades the entity lock operation.

The following examples will explain some of the aforementioned cascade operations using the following entities:

JAVA

```java
@Entity
public class Person {

    @Id
    private Long id;

    private String name;

    @OneToMany(mappedBy = "owner", cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public List<Phone> getPhones() {
        return phones;
    }

    public void addPhone(Phone phone) {
        this.phones.add( phone );
        phone.setOwner( this );
    }
}


@Entity
public class Phone {

    @Id
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person owner;

    public Long getId() {
        return id;
    }

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

```
    public String getNumber() {
        return number;
    }

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

    public Person getOwner() {
        return owner;
    }

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

5.13.1. CascadeType.PERSIST

The `CascadeType.PERSIST` allows us to persist a child entity along with the parent one.

Example 278. `CascadeType.PERSIST` *example*

JAVA

```
Person person = new Person();
person.setId( 1L );
person.setName( "John Doe" );

Phone phone = new Phone();
phone.setId( 1L );
phone.setNumber( "123-456-7890" );

person.addPhone( phone );

entityManager.persist( person );
```

SQL

```
INSERT INTO Person ( name, id )
VALUES ( 'John Doe', 1 )

INSERT INTO Phone ( `number`, person_id, id )
VALUE ( '123-456-7890', 1, 1 )
```

Even if just the `Person` parent entity was persisted, Hibernate has managed to cascade the persist operation to the associated `Phone` child entity as well.

5.13.2. CascadeType.MERGE

The `CascadeType.MERGE` allows us to merge a child entity along with the parent one.

Example 279. `CascadeType.MERGE` *example*

JAVA

```
Phone phone = entityManager.find( Phone.class, 1L );
Person person = phone.getOwner();

person.setName( "John Doe Jr." );
phone.setNumber( "987-654-3210" );

entityManager.clear();

entityManager.merge( person );
```

SQL

```
SELECT
    p.id as id1_0_1_,
    p.name as name2_0_1_,
    ph.owner_id as owner_id3_1_3_,
    ph.id as id1_1_3_,
    ph.id as id1_1_0_,
    ph."number" as number2_1_0_,
    ph.owner_id as owner_id3_1_0_
FROM
    Person p
LEFT OUTER JOIN
    Phone ph
        on p.id=ph.owner_id
WHERE
    p.id = 1
```

During merge, the current state of the entity is copied onto the entity version that was just fetched from the database. That's the reason why Hibernate executed the SELECT statement which fetched both the `Person` entity along with its children.

5.13.3. `CascadeType.REMOVE`

The `CascadeType.REMOVE` allows us to remove a child entity along with the parent one. Traditionally, Hibernate called this operation delete, that's why the `org.hibernate.annotations.CascadeType` provides a `DELETE` cascade option. However, `CascadeType.REMOVE` and `org.hibernate.annotations.CascadeType.DELETE` are identical.

Example 280. `CascadeType.REMOVE` *example*

JAVA

```
Person person = entityManager.find( Person.class, 1L );

entityManager.remove( person );
```

```
DELETE FROM Phone WHERE id = 1

DELETE FROM Person WHERE id = 1
```
SQL

5.13.4. CascadeType.DETACH

`CascadeType.DETACH` is used to propagate the detach operation from a parent entity to a child.

Example 281. `CascadeType.DETACH` *example*

```
Person person = entityManager.find( Person.class, 1L );
assertEquals( 1, person.getPhones().size() );
Phone phone = person.getPhones().get( 0 );

assertTrue( entityManager.contains( person ));
assertTrue( entityManager.contains( phone ));

entityManager.detach( person );

assertFalse( entityManager.contains( person ));
assertFalse( entityManager.contains( phone ));
```
JAVA

5.13.5. CascadeType.LOCK

Although unintuitively, `CascadeType.LOCK` does not propagate a lock request from a parent entity to its children. Such a use case requires the use of the `PessimisticLockScope.EXTENDED` value of the `javax.persistence.lock.scope` property.

However, `CascadeType.LOCK` allows us to reattach a parent entity along with its children to the currently running Persistence Context.

Example 282. `CascadeType.LOCK` *example*

```
Person person = entityManager.find( Person.class, 1L );
assertEquals( 1, person.getPhones().size() );
Phone phone = person.getPhones().get( 0 );

assertTrue( entityManager.contains( person ) );
assertTrue( entityManager.contains( phone ) );

entityManager.detach( person );

assertFalse( entityManager.contains( person ) );
assertFalse( entityManager.contains( phone ) );

entityManager.unwrap( Session.class )
        .buildLockRequest( new LockOptions( LockMode.NONE ) )
        .lock( person );

assertTrue( entityManager.contains( person ) );
assertTrue( entityManager.contains( phone ) );
```
JAVA

5.13.6. CascadeType.REFRESH

The `CascadeType.REFRESH` is used to propagate the refresh operation from a parent entity to a child. The refresh operation will discard the current entity state, and it will override it using the one loaded from the database.

Example 283. `CascadeType.REFRESH` *example*

```
Person person = entityManager.find( Person.class, 1L );
Phone phone = person.getPhones().get( 0 );

person.setName( "John Doe Jr." );
phone.setNumber( "987-654-3210" );

entityManager.refresh( person );

assertEquals( "John Doe", person.getName() );
assertEquals( "123-456-7890", phone.getNumber() );
```
JAVA

```
SQL
SELECT
    p.id as id1_0_1_,
    p.name as name2_0_1_,
    ph.owner_id as owner_id3_1_3_,
    ph.id as id1_1_3_,
    ph.id as id1_1_0_,
    ph."number" as number2_1_0_,
    ph.owner_id as owner_id3_1_0_
FROM
    Person p
LEFT OUTER JOIN
    Phone ph
        ON p.id=ph.owner_id
WHERE
    p.id = 1
```

In the aforementioned example, you can see that both the `Person` and `Phone` entities are refreshed even if we only called this operation on the parent entity only.

5.13.7. `CascadeType.REPLICATE`

The `CascadeType.REPLICATE` is to replicate both the parent and the child entities. The replicate operation allows you to synchronize entities coming from different sources of data.

Example 284. `CascadeType.REPLICATE` *example*

```
JAVA
Person person = new Person();
person.setId( 1L );
person.setName( "John Doe Sr." );

Phone phone = new Phone();
phone.setId( 1L );
phone.setNumber( "(01) 123-456-7890" );
person.addPhone( phone );

entityManager.unwrap( Session.class ).replicate( person, ReplicationMode.OVERWRITE );
```

```sql
SELECT
    id
FROM
    Person
WHERE
    id = 1

SELECT
    id
FROM
    Phone
WHERE
    id = 1

UPDATE
    Person
SET
    name = 'John Doe Sr.'
WHERE
    id = 1

UPDATE
    Phone
SET
    "number" = '(01) 123-456-7890',
    owner_id = 1
WHERE
    id = 1
```

As illustrated by the SQL statements being generated, both the `Person` and `Phone` entities are replicated to the underlying database rows.

6. Flushing

Flushing is the process of synchronizing the state of the persistence context with the underlying database. The `EntityManager` and the Hibernate `Session` expose a set of methods, through which the application developer can change the persistent state of an entity.

The persistence context acts as a transactional write-behind cache, queuing any entity state change. Like any write-behind cache, changes are first applied in-memory and synchronized with the database during flush time. The flush operation takes every entity state change and translates it to an `INSERT`, `UPDATE` or `DELETE` statement.

> Because DML statements are grouped together, Hibernate can apply batching transparently. See the Batching chapter for more information.

The flushing strategy is given by the `flushMode` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html#getFlushMode--) of the current running Hibernate `Session`. Although JPA defines only two flushing strategies (`AUTO` (https://docs.oracle.com/javaee/7/api/javax/persistence/FlushModeType.html#AUTO) and `COMMIT` (https://docs.oracle.com/javaee/7/api/javax/persistence/FlushModeType.html#COMMIT)), Hibernate has a much broader spectrum of flush types:

ALWAYS
: Flushes the `Session` before every query.

AUTO
: This is the default mode and it flushes the `Session` only if necessary.

COMMIT
: The `Session` tries to delay the flush until the current `Transaction` is committed, although it might flush prematurely too.

MANUAL
: The `Session` flushing is delegated to the application, which must call `Session.flush()` explicitly in order to apply the persistence context changes.

6.1. `AUTO` flush

By default, Hibernate uses the `AUTO` flush mode which triggers a flush in the following circumstances:

- prior to committing a `Transaction`
- prior to executing a JPQL/HQL query that overlaps with the queued entity actions
- before executing any native SQL query that has no registered synchronization

6.1.1. `AUTO` flush on commit

In the following example, an entity is persisted and then the transaction is committed.

Example 285. Automatic flushing on commit

```
JAVA
entityManager = entityManagerFactory().createEntityManager();
txn = entityManager.getTransaction();
txn.begin();

Person person = new Person( "John Doe" );
entityManager.persist( person );
log.info( "Entity is in persisted state" );

txn.commit();


SQL
--INFO: Entity is in persisted state
INSERT INTO Person (name, id) VALUES ('John Doe', 1)
```

Hibernate logs the message prior to inserting the entity because the flush only occurred during transaction commit.

> This is valid for the `SEQUENCE` and `TABLE` identifier generators. The `IDENTITY` generator must execute the insert right after calling `persist()`. For details, see the discussion of generators in *Identifier generators*.

6.1.2. `AUTO` flush on JPQL/HQL query

A flush may also be triggered when executing an entity query.

Example 286. Automatic flushing on JPQL/HQL

```
JAVA
Person person = new Person( "John Doe" );
entityManager.persist( person );
entityManager.createQuery( "select p from Advertisement p" ).getResultList();
entityManager.createQuery( "select p from Person p" ).getResultList();
```

```
SQL
SELECT a.id AS id1_0_ ,
       a.title AS title2_0_
FROM   Advertisement a

INSERT INTO Person (name, id) VALUES ('John Doe', 1)

SELECT p.id AS id1_1_ ,
       p.name AS name2_1_
FROM   Person p
```

The reason why the `Advertisement` entity query didn't trigger a flush is because there's no overlapping between the `Advertisement` and the `Person` tables:

Example 287. Automatic flushing on JPQL/HQL entities

JAVA

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

}

@Entity(name = "Advertisement")
public static class Advertisement {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

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

When querying for a `Person` entity, the flush is triggered prior to executing the entity query.

Example 288. Automatic flushing on JPQL/HQL

JAVA

```
Person person = new Person( "John Doe" );
entityManager.persist( person );
entityManager.createQuery( "select p from Person p" ).getResultList();
```

SQL

```
INSERT INTO Person (name, id) VALUES ('John Doe', 1)

SELECT p.id AS id1_1_ ,
       p.name AS name2_1_
FROM   Person p
```

This time, the flush was triggered by a JPQL query because the pending entity persist action overlaps with the query being executed.

6.1.3. `AUTO` flush on native SQL query

When executing a native SQL query, a flush is always triggered when using the `EntityManager` API.

Example 289. Automatic flushing on native SQL using `EntityManager`

JAVA

```
assertTrue(((Number) entityManager
        .createNativeQuery( "select count(*) from Person")
        .getSingleResult()).intValue() == 0 );

Person person = new Person( "John Doe" );
entityManager.persist( person );

assertTrue(((Number) entityManager
        .createNativeQuery( "select count(*) from Person")
        .getSingleResult()).intValue() == 1 );
```

The `Session` API doesn't trigger an `AUTO` flush when executing a native query

Example 290. Automatic flushing on native SQL using `Session`

JAVA

```
                assertTrue(((Number) entityManager
                        .createNativeQuery( "select count(*) from Person")
                        .getSingleResult()).intValue() == 0 );

                Person person = new Person( "John Doe" );
                entityManager.persist( person );
                Session session = entityManager.unwrap(Session.class);

                // for this to work, the Session/EntityManager must be put into COMMIT FlushMode
                //  - this is a change since 5.2 to account for merging EntityManager functionality
                //              directly into Session.  Flushing would be the JPA-spec compliant behavior,
                //              so we know do that by default.
                session.setFlushMode( FlushModeType.COMMIT );
                //              or using Hibernate's FlushMode enum
                //session.setHibernateFlushMode( FlushMode.COMMIT );

                assertTrue(((Number) session
                        .createSQLQuery( "select count(*) from Person")
                        .uniqueResult()).intValue() == 0 );
                //end::flushing-auto-flush-sql-native-example[\]
        } );
}

@Test
public void testFlushAutoSQLSynchronization() {
        doInJPA( this::entityManagerFactory, entityManager -> {
                entityManager.createNativeQuery( "delete from Person" ).executeUpdate();;
        } );
        doInJPA( this::entityManagerFactory, entityManager -> {
                log.info( "testFlushAutoSQLSynchronization" );
                assertTrue(((Number) entityManager
                        .createNativeQuery( "select count(*) from Person")
                        .getSingleResult()).intValue() == 0 );

                Person person = new Person( "John Doe" );
                entityManager.persist( person );
                Session session = entityManager.unwrap( Session.class );

                assertTrue(((Number) session
                        .createSQLQuery( "select count(*) from Person")
                        .addSynchronizedEntityClass( Person.class )
                        .uniqueResult()).intValue() == 1 );
        } );
}

@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        private String name;

        public Person() {}
```

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

                public Long getId() {
                        return id;
                }

                public String getName() {
                        return name;
                }

        }

        @Entity(name = "Advertisement")
        public static class Advertisement {

                @Id
                @GeneratedValue
                private Long id;

                private String title;

                public Long getId() {
                        return id;
                }

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

                public String getTitle() {
                        return title;
                }

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

To flush the `Session`, the query must use a synchronization:

Example 291. Automatic flushing on native SQL with `Session` synchronization

```
JAVA
assertTrue(((Number) entityManager
        .createNativeQuery( "select count(*) from Person")
        .getSingleResult()).intValue() == 0 );

Person person = new Person( "John Doe" );
entityManager.persist( person );
Session session = entityManager.unwrap( Session.class );

assertTrue(((Number) session
        .createSQLQuery( "select count(*) from Person")
        .addSynchronizedEntityClass( Person.class )
        .uniqueResult()).intValue() == 1 );
```

6.2. COMMIT flush

JPA also defines a COMMIT flush mode, which is described as follows:

> *If* `FlushModeType.COMMIT` *is set, the effect of updates made to entities in the persistence context upon queries is unspecified.*
>
> — *Section 3.10.8 of the JPA 2.1 Specification*

When executing a JPQL query, the persistence context is only flushed when the current running transaction is committed.

Example 292. `COMMIT` *flushing on JPQL*

```
JAVA
Person person = new Person("John Doe");
entityManager.persist(person);

entityManager.createQuery("select p from Advertisement p")
    .setFlushMode( FlushModeType.COMMIT)
    .getResultList();

entityManager.createQuery("select p from Person p")
    .setFlushMode( FlushModeType.COMMIT)
    .getResultList();
```

```
SQL
SELECT a.id AS id1_0_ ,
       a.title AS title2_0_
FROM   Advertisement a

SELECT p.id AS id1_1_ ,
       p.name AS name2_1_
FROM   Person p

INSERT INTO Person (name, id) VALUES ('John Doe', 1)
```

Because the JPA doesn't impose a strict rule on delaying flushing, when executing a native SQL query, the persistence context is going to be flushed.

Example 293. `COMMIT` *flushing on SQL*

JAVA

```
Person person = new Person("John Doe");
entityManager.persist(person);

assertTrue(((Number) entityManager
    .createNativeQuery("select count(*) from Person")
    .getSingleResult()).intValue() == 1);
```

SQL

```
INSERT INTO Person (name, id) VALUES ('John Doe', 1)

SELECT COUNT(*) FROM Person
```

6.3. `ALWAYS` flush

> The `ALWAYS` is only available with the native `Session` API.

The `ALWAYS` flush mode triggers a persistence context flush even when executing a native SQL query against the `Session` API.

Example 294. `COMMIT` *flushing on SQL*

JAVA

```
Person person = new Person("John Doe");
entityManager.persist(person);

Session session = entityManager.unwrap( Session.class);
assertTrue(((Number) session
        .createSQLQuery("select count(*) from Person")
        .setFlushMode( FlushMode.ALWAYS)
        .uniqueResult()).intValue() == 1);
```

```
INSERT INTO Person (name, id) VALUES ('John Doe', 1)

SELECT COUNT(*) FROM Person
```
SQL

6.4. MANUAL flush

Both the `EntityManager` and the Hibernate `Session` define a `flush()` method that, when called, triggers a manual flush. Hibernate also defines a `MANUAL` flush mode so the persistence context can only be flushed manually.

Example 295. `MANUAL` *flushing*

```
Person person = new Person("John Doe");
entityManager.persist(person);

Session session = entityManager.unwrap( Session.class);
session.setHibernateFlushMode( FlushMode.MANUAL );

assertTrue(((Number) entityManager
    .createQuery("select count(id) from Person")
    .getSingleResult()).intValue() == 0);

assertTrue(((Number) session
    .createSQLQuery("select count(*) from Person")
    .uniqueResult()).intValue() == 0);
```
JAVA

```
SELECT COUNT(p.id) AS col_0_0_
FROM   Person p

SELECT COUNT(*)
FROM   Person
```
SQL

The `INSERT` statement was not executed because the persistence context because there was no manual `flush()` call.

> This mode is useful when using multi-request logical transactions and only the last request should flush the persistence context.

6.5. Flush operation order

From a database perspective, a row state can be altered using either an `INSERT`, an `UPDATE` or a `DELETE` statement. Because entity state changes are automatically converted to SQL statements, it's important to know which entity actions are associated to a given SQL statement.

`INSERT`

The `INSERT` statement is generated either by the `EntityInsertAction` or `EntityIdentityInsertAction`. These actions are scheduled by the `persist` operation, either explicitly or through cascading the `PersistEvent` from a parent to a child entity.

`DELETE`

The `DELETE` statement is generated by the `EntityDeleteAction` or `OrphanRemovalAction`.

`UPDATE`

The `UPDATE` statement is generated by `EntityUpdateAction` during flushing if the managed entity has been marked modified. The dirty checking mechanism is responsible for determining if a managed entity has been modified since it was first loaded.

Hibernate does not execute the SQL statements in the order of their associated entity state operations.

To visualize how this works, consider the following example:

Example 296. Flush operation order

JAVA

```
Person person = entityManager.find( Person.class, 1L);
entityManager.remove(person);

Person newPerson = new Person( );
newPerson.setId( 2L );
newPerson.setName( "John Doe" );
entityManager.persist( newPerson );
```

SQL

```
INSERT INTO Person (name, id)
VALUES ('John Doe', 2L)

DELETE FROM Person WHERE id = 1
```

Even if we removed the first entity and then persist a new one, Hibernate is going to execute the `DELETE` statement after the `INSERT`.

> The order in which SQL statements are executed is given by the `ActionQueue` and not by the order in which entity state operations have been previously defined.

The `ActionQueue` executes all operations in the following order:

1. `OrphanRemovalAction`
2. `EntityInsertAction` or `EntityIdentityInsertAction`
3. `EntityUpdateAction`
4. `CollectionRemoveAction`
5. `CollectionUpdateAction`
6. `CollectionRecreateAction`
7. `EntityDeleteAction`

7. Database access

7.1. ConnectionProvider

As an ORM tool, probably the single most important thing you need to tell Hibernate is how to connect to your database so that it may connect on behalf of your application. This is ultimately the function of the `org.hibernate.engine.jdbc.connections.spi.ConnectionProvider` interface. Hibernate provides some out of the box implementations of this interface. `ConnectionProvider` is also an extension point so you can also use custom implementations from third parties or written yourself. The `ConnectionProvider` to use is defined by the `hibernate.connection.provider_class` setting. See the `org.hibernate.cfg.AvailableSettings#CONNECTION_PROVIDER` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/cfg/AvailableSettings.html#CONNECTION_PROVIDER)

Generally speaking, applications should not have to configure a `ConnectionProvider` explicitly if using one of the Hibernate-provided implementations. Hibernate will internally determine which `ConnectionProvider` to use based on the following algorithm:

1. If `hibernate.connection.provider_class` is set, it takes precedence

2. else if `hibernate.connection.datasource` is set → Using DataSources
3. else if any setting prefixed by `hibernate.c3p0.` is set → Using c3p0
4. else if any setting prefixed by `hibernate.proxool.` is set → Using Proxool
5. else if any setting prefixed by `hibernate.hikari.` is set → Using Hikari
6. else if `hibernate.connection.url` is set → Using Hibernate's built-in (and unsupported) pooling
7. else → User-provided Connections

7.2. Using DataSources

Hibernate can integrate with a `javax.sql.DataSource` for obtaining JDBC Connections. Applications would tell Hibernate about the `DataSource` via the (required) `hibernate.connection.datasource` setting which can either specify a JNDI name or would reference the actual `DataSource` instance. For cases where a JNDI name is given, be sure to read JNDI

> For JPA applications, note that `hibernate.connection.datasource` corresponds to either `javax.persistence.jtaDataSource` or `javax.persistence.nonJtaDataSource`.

The `DataSource` `ConnectionProvider` also (optionally) accepts the `hibernate.connection.username` and `hibernate.connection.password`. If specified, the `DataSource#getConnection(String username, String password)` (https://docs.oracle.com/javase/8/docs/api/javax/sql/DataSource.html#getConnection-java.lang.String-java.lang.String-) will be used. Otherwise, the no-arg form is used.

7.3. Using c3p0

> To use this integration, the application must include the hibernate-c3p0 module jar (as well as its dependencies) on the classpath.

Hibernate also provides support for applications to use c3p0 (http://www.mchange.com/projects/c3p0/) connection pooling. When using this c3p0 support, a number of additional configuration settings are recognized.

Transaction isolation of the Connections is managed by the `ConnectionProvider` itself. See ConnectionProvider support for transaction isolation setting.

`hibernate.connection.driver_class`
The name of the JDBC Driver class to use

`hibernate.connection.url`
The JDBC connection url.

Any settings prefixed with `hibernate.connection.` (other than the "special ones")
These all have the `hibernate.connection.` prefix stripped and the rest will be passed as JDBC connection properties

`hibernate.c3p0.min_size` or `c3p0.minPoolSize`
The minimum size of the c3p0 pool. See c3p0 minPoolSize (http://www.mchange.com/projects/c3p0/#minPoolSize)

`hibernate.c3p0.max_size` or `c3p0.maxPoolSize`
The maximum size of the c3p0 pool. See c3p0 maxPoolSize (http://www.mchange.com/projects/c3p0/#maxPoolSize)

`hibernate.c3p0.timeout` or `c3p0.maxIdleTime`
The Connection idle time. See c3p0 maxIdleTime (http://www.mchange.com/projects/c3p0/#maxIdleTime)

`hibernate.c3p0.max_statements` or `c3p0.maxStatements`
Controls the c3p0 PreparedStatement cache size (if using). See c3p0 maxStatements (http://www.mchange.com/projects/c3p0/#maxStatements)

`hibernate.c3p0.acquire_increment` or `c3p0.acquireIncrement`
Number of connections c3p0 should acquire at a time when pool is exhausted. See c3p0 acquireIncrement (http://www.mchange.com/projects/c3p0/#acquireIncrement)

`hibernate.c3p0.idle_test_period` or `c3p0.idleConnectionTestPeriod`
Idle time before a c3p0 pooled connection is validated. See c3p0 idleConnectionTestPeriod (http://www.mchange.com/projects/c3p0/#idleConnectionTestPeriod)

`hibernate.c3p0.initialPoolSize`

The initial c3p0 pool size. If not specified, default is to use the min pool size. See c3p0 initialPoolSize (http://www.mchange.com/projects/c3p0/#initialPoolSize)

Any other settings prefixed with `hibernate.c3p0.`

Will have the `hibernate.` portion stripped and be passed to c3p0.

Any other settings prefixed with `c3p0.`

Get passed to c3p0 as is. See c3p0 configuration (http://www.mchange.com/projects/c3p0/#configuration)

7.4. Using Proxool

> To use this integration, the application must include the hibernate-proxool module jar (as well as its dependencies) on the classpath.

Hibernate also provides support for applications to use Proxool (http://proxool.sourceforge.net/) connection pooling.

Transaction isolation of the Connections is managed by the `ConnectionProvider` itself. See ConnectionProvider support for transaction isolation setting.

7.5. Using existing Proxool pools

Controlled by the `hibernate.proxool.existing_pool` setting. If set to true, this ConnectionProvider will use an already existing Proxool pool by alias as indicated by the `hibernate.proxool.pool_alias` setting.

7.6. Configuring Proxool via XML

The `hibernate.proxool.xml` setting names a Proxool configuration XML file to be loaded as a classpath resource and loaded by Proxool's JAXPConfigurator. See proxool configuration (http://proxool.sourceforge.net/configure.html). `hibernate.proxool.pool_alias` must be set to indicate which pool to use.

7.7. Configuring Proxool via Properties

The `hibernate.proxool.properties` setting names a Proxool configuration properties file to be loaded as a classpath resource and loaded by Proxool's `PropertyConfigurator`. See proxool configuration (http://proxool.sourceforge.net/configure.html). `hibernate.proxool.pool_alias` must be set to indicate which pool to use.

7.8. Using Hikari

> To use this integration, the application must include the hibernate-hikari module jar (as well as its dependencies) on the classpath.

Hibernate also provides support for applications to use Hikari (http://brettwooldridge.github.io/HikariCP/) connection pool.

Set all of your Hikari settings in Hibernate prefixed by `hibernate.hikari.` and this `ConnectionProvider` will pick them up and pass them along to Hikari. Additionally, this `ConnectionProvider` will pick up the following Hibernate-specific properties and map them to the corresponding Hikari ones (any `hibernate.hikari.` prefixed ones have precedence):

`hibernate.connection.driver_class`
Mapped to Hikari's `driverClassName` setting

`hibernate.connection.url`
Mapped to Hikari's `jdbcUrl` setting

`hibernate.connection.username`
Mapped to Hikari's `username` setting

`hibernate.connection.password`
Mapped to Hikari's `password` setting

`hibernate.connection.isolation`
Mapped to Hikari's `transactionIsolation` setting. See ConnectionProvider support for transaction isolation setting. Note that Hikari only supports JDBC standard isolation levels (apparently).

`hibernate.connection.autocommit`
Mapped to Hikari's `autoCommit` setting

7.9. Using Hibernate's built-in (and unsupported) pooling

> The built-in connection pool is not supported supported for use.

This section is here just for completeness.

7.10. User-provided Connections

It is possible to use Hibernate by simply passing a Connection to use to the Session when the Session is opened. This usage is discouraged and not discussed here.

7.11. ConnectionProvider support for transaction isolation setting

All of the provided ConnectionProvider implementations, other than `DataSourceConnectionProvider`, support consistent setting of transaction isolation for all `Connections` obtained from the underlying pool. The value for `hibernate.connection.isolation` can be specified in one of 3 formats:

- the integer value accepted at the JDBC level
- the name of the `java.sql.Connection` constant field representing the isolation you would like to use. For example, `TRANSACTION_REPEATABLE_READ` for `java.sql.Connection#TRANSACTION_REPEATABLE_READ` (https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#TRANSACTION_REPEATABLE_READ). Not that this is only supported for JDBC standard isolation levels, not for isolation levels specific to a particular JDBC driver.
- a short-name version of the java.sql.Connection constant field without the `TRANSACTION_` prefix. For example, `REPEATABLE_READ` for `java.sql.Connection#TRANSACTION_REPEATABLE_READ` (https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#TRANSACTION_REPEATABLE_READ). Again, this is only supported for JDBC standard isolation levels, not for isolation levels specific to a particular JDBC driver.

7.12. Database Dialect

Although SQL is relatively standardized, each database vendor uses a subset and superset of ANSI SQL defined syntax. This is referred to as the database's dialect. Hibernate handles variations across these dialects through its `org.hibernate.dialect.Dialect` class and the various subclasses for each database vendor.

In most cases Hibernate will be able to determine the proper Dialect to use by asking some questions of the JDBC Connection during bootstrap. For information on Hibernate's ability to determine the proper Dialect to use (and your ability to influence that resolution), see Dialect resolution.

If for some reason it is not able to determine the proper one or you want to use a custom Dialect, you will need to set the `hibernate.dialect` setting.

Table 4. Provided Dialects

Dialect (short name)	Remarks
Cache71	Support for the Caché database, version 2007.1
CUBRID	Support for the CUBRID database, version 8.3. May work with later versions.
DB2	Support for the DB2 database
DB2390	Support for DB2 Universal Database for OS/390, also known as DB2/390.
DB2400	Support for DB2 Universal Database for iSeries, also known as DB2/400.
DerbyTenFive	Support for the Derby database, version 10.5
DerbyTenSix	Support for the Derby database, version 10.6
DerbyTenSeven	Support for the Derby database, version 10.7
Firebird	Support for the Firebird database
FrontBase	Support for the Frontbase database
H2	Support for the H2 database
HSQL	Support for the HSQL (HyperSQL) database
Informix	Support for the Informix database
Ingres	Support for the Ingres database, version 9.2
Ingres9	Support for the Ingres database, version 9.3. May work with newer versions
Ingres10	Support for the Ingres database, version 10. May work with newer versions
Interbase	Support for the Interbase database.
JDataStore	Support for the JDataStore database
McKoi	Support for the McKoi database
Mimer	Support for the Mimer database, version 9.2.1. May work with newer versions

Dialect (short name)	Remarks
MySQL5	Support for the MySQL database, version 5.x
MySQL5InnoDB	Support for the MySQL database, version 5.x preferring the InnoDB storage engine when exporting tables.
MySQL57InnoDB	Support for the MySQL database, version 5.7 preferring the InnoDB storage engine when exporting tables. May work with newer versions
MariaDB	Support for the MariadB database. May work with newer versions
MariaDB53	Support for the MariadB database, version 5.3 and newer.
Oracle8i	Support for the Oracle database, version 8i
Oracle9i	Support for the Oracle database, version 9i
Oracle10g	Support for the Oracle database, version 10g
Pointbase	Support for the Pointbase database
PostgresPlus	Support for the Postgres Plus database
PostgreSQL81	Support for the PostgrSQL database, version 8.1
PostgreSQL82	Support for the PostgreSQL database, version 8.2
PostgreSQL9	Support for the PostgreSQL database, version 9. May work with later versions.
Progress	Support for the Progress database, version 9.1C. May work with newer versions.
SAPDB	Support for the SAPDB/MAXDB database.
SQLServer	Support for the SQL Server 2000 database
SQLServer2005	Support for the SQL Server 2005 database
SQLServer2008	Support for the SQL Server 2008 database
Sybase11	Support for the Sybase database, up to version 11.9.2
SybaseAnywhere	Support for the Sybase Anywhere database
SybaseASE15	Support for the Sybase Adaptive Server Enterprise database, version 15

Dialect (short name)	Remarks
SybaseASE157	Support for the Sybase Adaptive Server Enterprise database, version 15.7. May work with newer versions.
Teradata	Support for the Teradata database
TimesTen	Support for the TimesTen database, version 5.1. May work with newer versions

8. Transactions and concurrency control

It is important to understand that the term transaction has many different yet related meanings in regards to persistence and Object/Relational Mapping. In most use-cases these definitions align, but that is not always the case.

- Might refer to the physical transaction with the database.
- Might refer to the logical notion of a transaction as related to a persistence context.
- Might refer to the application notion of a Unit-of-Work, as defined by the archetypal pattern.

> This documentation largely treats the physical and logic notions of a transaction as one-in-the-same.

8.1. Physical Transactions

Hibernate uses the JDBC API for persistence. In the world of Java there are two well-defined mechanism for dealing with transactions in JDBC: JDBC itself and JTA. Hibernate supports both mechanisms for integrating with transactions and allowing applications to manage physical transactions.

Transaction handling per `Session` is handled by the `org.hibernate.resource.transaction.spi.TransactionCoordinator` contract, which are built by the `org.hibernate.resource.transaction.spi.TransactionCoordinatorBuilder` service. `TransactionCoordinatorBuilder` represents a strategy for dealing with transactions whereas TransactionCoordinator

represents one instance of that strategy related to a Session. Which `TransactionCoordinatorBuilder` implementation to use is defined by the `hibernate.transaction.coordinator_class` setting.

`jdbc` (the default for non-JPA applications)
: Manages transactions via calls to `java.sql.Connection`

`jta`
: Manages transactions via JTA. See Java EE bootstrapping

If a JPA application does not provide a setting for `hibernate.transaction.coordinator_class`, Hibernate will automatically build the proper transaction coordinator based on the transaction type for the persistence unit.

If a non-JPA application does not provide a setting for `hibernate.transaction.coordinator_class`, Hibernate will use `jdbc` as the default. This default will cause problems if the application actually uses JTA-based transactions. A non-JPA application that uses JTA-based transactions should explicitly set `hibernate.transaction.coordinator_class=jta` or provide a custom `org.hibernate.resource.transaction.TransactionCoordinatorBuilder` that builds a `org.hibernate.resource.transaction.TransactionCoordinator` that properly coordinates with JTA-based transactions.

> For details on implementing a custom `TransactionCoordinatorBuilder`, or simply better understanding how it works, see the Integrations Guide.

Hibernate uses JDBC connections and JTA resources directly, without adding any additional locking behavior. Hibernate does not lock objects in memory. The behavior defined by the isolation level of your database transactions does not change when you use Hibernate. The Hibernate `Session` acts as a transaction-scoped cache providing repeatable reads for lookup by identifier and queries that result in loading entities.

> To reduce lock contention in the database, the physical database transaction needs to be as short as possible. Long database transactions prevent your application from scaling to a highly-concurrent load. Do not hold a database transaction open during end-user-level work, but open it after the end-user-level work is finished. This is concept is referred to as `transactional write-behind`.

8.2. JTA configuration

Interaction with a JTA system is consolidated behind a single contract named `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform` which exposes access to the `javax.transaction.TransactionManager` and `javax.transaction.UserTransaction` for that system as well as exposing the ability to register `javax.transaction.Synchronization` instances, check transaction status, etc.

> Generally, `JtaPlatform` will need access to JNDI to resolve the JTA `TransactionManager`, `UserTransaction`, etc. See JNDI chapter for details on configuring access to JNDI.

Hibernate tries to discover the `JtaPlatform` it should use through the use of another service named `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatformResolver`. If that resolution does not work, or if you wish to provide a custom implementation you will need to specify the `hibernate.transaction.jta.platform` setting. Hibernate provides many implementations of the `JtaPlatform` contract, all with short names:

`Borland`
: `JtaPlatform` for the Borland Enterprise Server.

`Bitronix`
: `JtaPlatform` for Bitronix.

`JBossAS`
: `JtaPlatform` for Arjuna/JBossTransactions/Narayana when used within the JBoss/WildFly Application Server.

`JBossTS`
: `JtaPlatform` for Arjuna/JBossTransactions/Narayana when used standalone.

`JOnAS`
: `JtaPlatform` for JOTM when used within JOnAS.

`JOTM`

`JtaPlatform` for JOTM when used standalone.

`JRun4`

`JtaPlatform` for the JRun 4 Application Server.

`OC4J`

`JtaPlatform` for Oracle's OC4J container.

`Orion`

`JtaPlatform` for the Orion Application Server.

`Resin`

`JtaPlatform` for the Resin Application Server.

`SunOne`

`JtaPlatform` for the SunOne Application Server.

`Weblogic`

`JtaPlatform` for the Weblogic Application Server.

`WebSphere`

`JtaPlatform` for older versions of the WebSphere Application Server.

`WebSphereExtended`

`JtaPlatform` for newer versions of the WebSphere Application Server.

8.3. Hibernate Transaction API

Hibernate provides an API for helping to isolate applications from the differences in the underlying physical transaction system in use. Based on the configured `TransactionCoordinatorBuilder`, Hibernate will simply do the right thing when this transaction API is used by the application. This allows your applications and components to be more portable move around into different environments.

To use this API, you would obtain the `org.hibernate.Transaction` from the Session. `Transaction` allows for all the normal operations you'd expect: `begin`, `commit` and `rollback`, and it even exposes some cool methods like:

`markRollbackOnly`

that works in both JTA and JDBC

`getTimeout and setTimeout`

that again work in both JTA and JDBC

`registerSynchronization`

that allows you to register JTA Synchronizations even in non-JTA environments. In fact in both JTA and JDBC environments, these `Synchronizations` are kept locally by Hibernate. In JTA environments, Hibernate will only ever register one single `Synchronization` with the `TransactionManager` to avoid ordering problems.

Additionally, it exposes a getStatus method that returns an `org.hibernate.resource.transaction.spi.TransactionStatus` enum. This method checks with the underlying transaction system if needed, so care should be taken to minimize its use; it can have a big performance impact in certain JTA set ups.

Let's take a look at using the Transaction API in the various environments.

Example 297. Using Transaction API in JDBC

JAVA

```
StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
        // "jdbc" is the default, but for explicitness
        .applySetting( AvailableSettings.TRANSACTION_COORDINATOR_STRATEGY, "jdbc" )
        .build();

Metadata metadata = new MetadataSources( serviceRegistry )
        .addAnnotatedClass( Customer.class )
        .getMetadataBuilder()
        .build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
        .build();

Session session = sessionFactory.openSession();
try {
    // calls Connection#setAutoCommit( false ) to
    // signal start of transaction
    session.getTransaction().begin();

    session.createQuery( "UPDATE customer set NAME = 'Sir. '||NAME" )
            .executeUpdate();

    // calls Connection#commit(), if an error
    // happens we attempt a rollback
    session.getTransaction().commit();
}
catch ( Exception e ) {
    // we may need to rollback depending on
    // where the exception happened
    if ( session.getTransaction().getStatus() == TransactionStatus.ACTIVE
            || session.getTransaction().getStatus() == TransactionStatus.MARKED_ROLLBACK ) {
        session.getTransaction().rollback();
    }
    // handle the underlying error
}
finally {
    session.close();
    sessionFactory.close();
}
```

Example 298. Using Transaction API in JTA (CMT)

JAVA

```
StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
        // "jdbc" is the default, but for explicitness
        .applySetting( AvailableSettings.TRANSACTION_COORDINATOR_STRATEGY, "jta" )
        .build();

Metadata metadata = new MetadataSources( serviceRegistry )
        .addAnnotatedClass( Customer.class )
        .getMetadataBuilder()
        .build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
        .build();

// Note: depending on the JtaPlatform used and some optional settings,
// the underlying transactions here will be controlled through either
// the JTA TransactionManager or UserTransaction

Session session = sessionFactory.openSession();
try {
    // Since we are in CMT, a JTA transaction would
    // already have been started.  This call essentially
    // no-ops
    session.getTransaction().begin();

    Number customerCount = (Number) session.createQuery( "select count(c) from Customer c" ).uniqueResult();

    // Since we did not start the transaction ( CMT ),
    // we also will not end it.  This call essentially
    // no-ops in terms of transaction handling.
    session.getTransaction().commit();
}
catch ( Exception e ) {
    // again, the rollback call here would no-op (aside from
    // marking the underlying CMT transaction for rollback only).
    if ( session.getTransaction().getStatus() == TransactionStatus.ACTIVE
            || session.getTransaction().getStatus() == TransactionStatus.MARKED_ROLLBACK ) {
        session.getTransaction().rollback();
    }
    // handle the underlying error
}
finally {
    session.close();
    sessionFactory.close();
}
```

Example 299. Using Transaction API in JTA (BMT)

JAVA

```
StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
        // "jdbc" is the default, but for explicitness
        .applySetting( AvailableSettings.TRANSACTION_COORDINATOR_STRATEGY, "jta" )
        .build();

Metadata metadata = new MetadataSources( serviceRegistry )
        .addAnnotatedClass( Customer.class )
        .getMetadataBuilder()
        .build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
        .build();

// Note: depending on the JtaPlatform used and some optional settings,
// the underlying transactions here will be controlled through either
// the JTA TransactionManager or UserTransaction

Session session = sessionFactory.openSession();
try {
    // Assuming a JTA transaction is not already active,
    // this call the TM/UT begin method.  If a JTA
    // transaction is already active, we remember that
    // the Transaction associated with the Session did
    // not "initiate" the JTA transaction and will later
    // nop-op the commit and rollback calls...
    session.getTransaction().begin();

    session.persist( new Customer(  ) );
    Customer customer = (Customer) session.createQuery( "select c from Customer c" ).uniqueResult();

    // calls TM/UT commit method, assuming we are initiator.
    session.getTransaction().commit();
}
catch ( Exception e ) {
    // we may need to rollback depending on
    // where the exception happened
    if ( session.getTransaction().getStatus() == TransactionStatus.ACTIVE
            || session.getTransaction().getStatus() == TransactionStatus.MARKED_ROLLBACK ) {
        // calls TM/UT commit method, assuming we are initiator;
        // otherwise marks the JTA transaction for rollback only
        session.getTransaction().rollback();
    }
    // handle the underlying error
}
finally {
    session.close();
    sessionFactory.close();
}
```

In the CMT case we really could have omitted all of the Transaction calls. But the point of the examples was to show that the Transaction API really does insulate your code from the underlying transaction mechanism. In fact, if you strip away the comments and the single configuration setting supplied at bootstrap, the code is exactly the same in all 3 examples. In other

words, we could develop that code and drop it, as-is, in any of the 3 transaction environments.

The Transaction API tries hard to make the experience consistent across all environments. To that end, it generally defers to the JTA specification when there are differences (for example automatically trying rollback on a failed commit).

8.4. Contextual sessions

Most applications using Hibernate need some form of *contextual* session, where a given session is in effect throughout the scope of a given context. However, across applications the definition of what constitutes a context is typically different; different contexts define different scopes to the notion of current. Applications using Hibernate prior to version 3.0 tended to utilize either home-grown `ThreadLocal` -based contextual sessions, helper classes such as `HibernateUtil` , or utilized third-party frameworks, such as Spring or Pico, which provided proxy/interception-based contextual sessions.

Starting with version 3.0.1, Hibernate added the `SessionFactory.getCurrentSession()` method. Initially, this assumed usage of `JTA` transactions, where the `JTA` transaction defined both the scope and context of a current session. Given the maturity of the numerous stand-alone `JTA TransactionManager` implementations, most, if not all, applications should be using `JTA` transaction management, whether or not they are deployed into a `J2EE` container. Based on that, the `JTA` -based contextual sessions are all you need to use.

However, as of version 3.1, the processing behind `SessionFactory.getCurrentSession()` is now pluggable. To that end, a new extension interface, `org.hibernate.context.spi.CurrentSessionContext` , and a new configuration parameter, `hibernate.current_session_context_class` , have been added to allow pluggability of the scope and context of defining current sessions.

See the Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/spi/CurrentSessionContext.html) for the `org.hibernate.context.spi.CurrentSessionContext` interface for a detailed discussion of its contract. It defines a single method, `currentSession()` , by which the implementation is responsible for tracking the current contextual session. Out-of-the-box, Hibernate comes with three implementations of this interface:

`org.hibernate.context.internal.JTASessionContext`

current sessions are tracked and scoped by a `JTA` transaction. The processing here is exactly the same as in the older JTA-only approach. See the Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/internal/JTASessionContext.html) for more details.

- `org.hibernate.context.internal.ThreadLocalSessionContext` :current sessions are tracked by thread of execution. See the Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/internal/ThreadLocalSessionContext.html) for more details.
- `org.hibernate.context.internal.ManagedSessionContext` : current sessions are tracked by thread of execution. However, you are responsible to bind and unbind a `Session` instance with static methods on this class: it does not open, flush, or close a `Session` . See the Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/internal/ManagedSessionContext.html) for details.

Typically, the value of this parameter would just name the implementation class to use. For the three out-of-the-box implementations, however, there are three corresponding short names: *jta*, *thread*, and *managed*.

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

The `hibernate.current_session_context_class` configuration parameter defines which `org.hibernate.context.spi.CurrentSessionContext` implementation should be used. For backwards compatibility, if this configuration parameter is not set but a `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform` is configured, Hibernate will use the `org.hibernate.context.internal.JTASessionContext`.

8.5. Transactional patterns (and anti-patterns)

8.6. Session-per-operation anti-pattern

This is an anti-pattern of opening and closing a `Session` for each database call in a single thread. It is also an anti-pattern in terms of database transactions. Group your database calls into a planned sequence. In the same way, do not auto-commit after every SQL statement in your application. Hibernate disables, or expects the application server to disable, auto-commit mode immediately. Database transactions are never optional. All communication with a database must be encapsulated by a transaction. Avoid auto-commit behavior for reading data because many small transactions are unlikely to perform better than one clearly-defined unit of work, and are more difficult to maintain and extend.

> Using auto-commit does not circumvent database transactions. Instead, when in auto-commit mode, JDBC drivers simply perform each call in an implicit transaction call. It is as if your application called commit after each and every JDBC call.

8.7. Session-per-request pattern

This is the most common transaction pattern. The term request here relates to the concept of a system that reacts to a series of requests from a client/user. Web applications are a prime example of this type of system, though certainly not the only one. At the beginning of handling such a request, the application opens a Hibernate Session, starts a transaction, performs all data related

work, ends the transaction and closes the Session. The crux of the pattern is the one-to-one relationship between the transaction and the Session.

Within this pattern there is a common technique of defining a current session to simplify the need of passing this `Session` around to all the application components that may need access to it. Hibernate provides support for this technique through the `getCurrentSession` method of the `SessionFactory`. The concept of a *current* session has to have a scope that defines the bounds in which the notion of *current* is valid. This is the purpose of the `org.hibernate.context.spi.CurrentSessionContext` contract.

There are 2 reliable defining scopes:

- First is a JTA transaction because it allows a callback hook to know when it is ending, which gives Hibernate a chance to close the `Session` and clean up. This is represented by the `org.hibernate.context.internal.JTASessionContext` implementation of the `org.hibernate.context.spi.CurrentSessionContext` contract. Using this implementation, a `Session` will be opened the first time `getCurrentSession` is called within that transaction.
- Secondly is this application request cycle itself. This is best represented with the `org.hibernate.context.internal.ManagedSessionContext` implementation of the `org.hibernate.context.spi.CurrentSessionContext` contract. Here an external component is responsible for managing the lifecycle and scoping of a *current* session. At the start of such a scope, `ManagedSessionContext#bind()` method is called passing in the `Session`. At the end, its `unbind()` method is called. Some common examples of such *external components* include:
  - `javax.servlet.Filter` implementation
  - AOP interceptor with a pointcut on the service methods
  - A proxy/interception container

> The `getCurrentSession()` method has one downside in a JTA environment. If you use it, `after_statement` connection release mode is also used by default. Due to a limitation of the JTA specification, Hibernate cannot automatically clean up any unclosed `ScrollableResults` or `Iterator` instances returned by `scroll()` or `iterate()`. Release the underlying database cursor by calling `ScrollableResults#close()` or `Hibernate.close(Iterator)` explicitly from a finally block.

8.8. Conversations

The session-per-request pattern is not the only valid way of designing units of work. Many business processes require a whole series of interactions with the user that are interleaved with database accesses. In web and enterprise applications, it is not acceptable for a database transaction to span a user interaction. Consider the following example:

The first screen of a dialog opens. The data seen by the user is loaded in a particular `Session` and database transaction. The user is free to modify the objects.

The user uses a UI element to save their work after five minutes of editing. The modifications are made persistent. The user also expects to have exclusive access to the data during the edit session.

Even though we have multiple databases access here, from the point of view of the user, this series of steps represents a single unit of work. There are many ways to implement this in your application.

A first naive implementation might keep the `Session` and database transaction open while the user is editing, using database-level locks to prevent other users from modifying the same data and to guarantee isolation and atomicity. This is an anti-pattern because lock contention is a bottleneck which will prevent scalability in the future.

Several database transactions are used to implement the conversation. In this case, maintaining isolation of business processes becomes the partial responsibility of the application tier. A single conversation usually spans several database transactions. These multiple database accesses can only be atomic as a whole if only one of these database transactions (typically the last one) stores the updated data. All others only read data. A common way to receive this data is through a wizard-style dialog spanning several request/response cycles. Hibernate includes some features which make this easy to implement.

Automatic Versioning	Hibernate can perform automatic optimistic concurrency control for you. It can automatically detect (at the end of the conversation) if a concurrent modification occurred during user think time.
Detached Objects	If you decide to use the session-per-request pattern, all loaded instances will be in the detached state during user think time. Hibernate allows you to reattach the objects and persist the modifications. The pattern is called session-per-request-with-detached-objects. Automatic versioning is used to isolate concurrent modifications.
Extended `Session`	The Hibernate `Session` can be disconnected from the underlying JDBC connection after the database transaction has been committed and reconnected when a new client request occurs. This pattern is known as session-per-conversation and makes even reattachment unnecessary. Automatic versioning is used to isolate concurrent modifications and the `Session` will not be allowed to flush automatically, only explicitly.

Session-per-request-with-detached-objects and session-per-conversation each have advantages and disadvantages.

8.9. Session-per-application

The *session-per-application* is also considered an anti-pattern. The Hibernate `Session`, like the JPA `EntityManager`, is not a thread-safe object and it is intended to be confined to a single thread at once. If the `Session` is shared among multiple threads, there will be race conditions as well as visibility issues, so beware of this.

An exception thrown by Hibernate means you have to rollback your database transaction and close the `Session` immediately. If your `Session` is bound to the application, you have to stop the application. Rolling back the database transaction does not put your business objects back into the state they were at the start of the transaction. This means that the database state and the business objects will be out of sync. Usually, this is not a problem because exceptions are not recoverable and you will have to start over after rollback anyway.

The `Session` caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an `OutOfMemoryException`. One solution is to call `clear()` and `evict()` to manage the `Session` cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in the Batching chapter. Keeping a `Session` open for the duration of a user session also means a higher probability of stale data.

9. JNDI

Hibernate does optionally interact with JNDI on the application's behalf. Generally, it does this when the application:

- has asked the SessionFactory be bound to JNDI
- has specified a DataSource to use by JNDI name
- is using JTA transactions and the `JtaPlatform` needs to do JNDI lookups for `TransactionManager`, `UserTransaction`, etc

All of these JNDI calls route through a single service whose role is `org.hibernate.engine.jndi.spi.JndiService`. The standard `JndiService` accepts a number of configuration settings

`hibernate.jndi.class`

names the javax.naming.InitialContext implementation class to use. See `javax.naming.Context#INITIAL_CONTEXT_FACTORY` (https://docs.oracle.com/javase/8/docs/api/javax/naming/Context.html#INITIAL_CONTEXT_FACTORY)

`hibernate.jndi.url`

names the JNDI InitialContext connection url. See `javax.naming.Context.PROVIDER_URL` (https://docs.oracle.com/javase/8/docs/api/javax/naming/Context.html#PROVIDER_URL)

Any other settings prefixed with `hibernate.jndi.` will be collected and passed along to the JNDI provider.

> The standard `JndiService` assumes that all JNDI calls are relative to the same `InitialContext`. If your application uses multiple naming servers for whatever reason, you will need a custom `JndiService` implementation to handle those details.

10. Locking

In a relational database, locking refers to actions taken to prevent data from changing between the time it is read and the time is used.

Your locking strategy can be either optimistic or pessimistic.

Optimistic

Optimistic locking (http://en.wikipedia.org/wiki/Optimistic_locking) assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

Pessimistic

Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Hibernate provides mechanisms for implementing both types of locking in your applications.

10.1. Optimistic

When your application uses long transactions or conversations that span several database transactions, you can store versioning data so that if the same entity is updated by two conversations, the last to commit changes is informed of the conflict, and does not override the other conversation's work. This approach guarantees some isolation, but scales well and works particularly well in *read-often-write-sometimes* situations.

Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.

> A version or timestamp property can never be null for a detached instance. Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys.

10.2. Dedicated version number

The version number mechanism for optimistic locking is provided through a `@Version` annotation.

Example 300. @Version annotation

```
@Version
private long version;
```

JAVA

Here, the version property is mapped to the `version` column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would otherwise be overwritten by a last-commit-wins strategy.

The version column can be any kind of type, as long as you define and implement the appropriate `UserVersionType`.

Your application is forbidden from altering the version number set by Hibernate. To artificially increase the version number, see the documentation for properties `LockModeType.OPTIMISTIC_FORCE_INCREMENT` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT` check in the Hibernate Entity Manager reference documentation.

> If the version number is generated by the database, such as a trigger, use the annotation `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)` on the version attribute.

10.3. Timestamp

Timestamps are a less reliable way of optimistic locking than version numbers, but can be used by applications for other purposes as well. Timestamping is automatically used if you the `@Version` annotation on a `Date` or `Calendar` property type.

Example 301. Using timestamps for optimistic locking

```
@Version
private Date version;
```
JAVA

Hibernate can retrieve the timestamp value from the database or the JVM, by reading the value you specify for the `@org.hibernate.annotations.Source` annotation. The value can be either `org.hibernate.annotations.SourceType.DB` or `org.hibernate.annotations.SourceType.VM`. The default behavior is to use the database, and is also used if you don't specify the annotation at all.

The timestamp can also be generated by the database instead of Hibernate, if you use the `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)` annotation.

10.4. Pessimistic

Typically, you only need to specify an isolation level for the JDBC connections and let the database handle locking issues. If you do need to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction, Hibernate gives you the tools you need.

> Hibernate always uses the locking mechanism of the database, and never lock objects in memory.

10.5. `LockMode` and `LockModeType`

Long before JPA 1.0, Hibernate already defined various explicit locking strategies through its `LockMode` enumeration. JPA comes with its own `LockModeType` (http://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html) enumeration which defines similar strategies as the Hibernate-native `LockMode`.

LockModeType	LockMode	**Description**

NONE	NONE	The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to `update()` or `saveOrUpdate()` also start out in this lock mode.
READ and OPTIMISTIC	READ	The entity version is checked towards the end of the currently running transaction.
WRITE and OPTIMISTIC_FORCE_INCREMENT	WRITE	The entity version is incremented automatically even if the entity has not changed.
PESSIMISTIC_FORCE_INCREMENT	PESSIMISTIC_FORCE_INCREMENT	The entity is locked pessimistically and its version is incremented automatically even if the entity has not changed.
PESSIMISTIC_READ	PESSIMISTIC_READ	The entity is locked pessimistically using a shared lock, if the database supports such a feature. Otherwise, an explicit lock is used.
PESSIMISTIC_WRITE	PESSIMISTIC_WRITE , UPGRADE	The entity is locked using an explicit lock.
PESSIMISTIC_WRITE with a javax.persistence.lock.timeout setting of 0	UPGRADE_NOWAIT	The lock acquisition request fails fast if the row s already locked.
PESSIMISTIC_WRITE with a javax.persistence.lock.timeout setting of -2	UPGRADE_SKIPLOCKED	The lock acquisition request skips the already locked rows. It uses a `SELECT … FOR UPDATE SKIP LOCKED` in Oracle and PostgreSQL 9.5, or `SELECT … with (rowlock, updlock, readpast)` in SQL Server .

The explicit user request mentioned above occurs as a consequence of any of the following actions:

- a call to `Session.load()` , specifying a `LockMode` .
- a call to `Session.lock()` .
- a call to `Query.setLockMode()` .

If you call `Session.load()` with option `UPGRADE` , `UPGRADE_NOWAIT` or `UPGRADE_SKIPLOCKED` , and the requested object is not already loaded by the session, the object is loaded using `SELECT … FOR UPDATE` .

If you call `load()` for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls `lock()` for that object.

`Session.lock()` performs a version number check if the specified lock mode is `READ`, `UPGRADE`, `UPGRADE_NOWAIT` or `UPGRADE_SKIPLOCKED`. In the case of `UPGRADE`, `UPGRADE_NOWAIT` or `UPGRADE_SKIPLOCKED`, the `SELECT … FOR UPDATE` syntax is used.

If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.

10.6. JPA locking query hints

JPA 2.0 introduced two query hints:

javax.persistence.lock.timeout
: it gives the number of milliseconds a lock acquisition request will wait before throwing an exception

javax.persistence.lock.scope
: defines the *scope* (http://docs.oracle.com/javaee/7/api/javax/persistence/PessimisticLockScope.html) of the lock acquisition request. The scope can either be `NORMAL` (default value) or `EXTENDED`. The `EXTENDED` scope will cause a lock acquisition request to be passed to other owned table structured (e.g. `@Inheritance(strategy=InheritanceType.JOINED)`, `@ElementCollection`)

Example 302. `javax.persistence.lock.timeout` *example*

JAVA
```
entityManager.find(
    Person.class, id, LockModeType.PESSIMISTIC_WRITE,
    Collections.singletonMap( "javax.persistence.lock.timeout", 200 )
);
```

SQL
```
SELECT explicitlo0_.id     AS id1_0_0_,
       explicitlo0_."name" AS name2_0_0_
FROM   person explicitlo0_
WHERE  explicitlo0_.id = 1
FOR UPDATE wait 2
```

> Not all JDBC database drivers support setting a timeout value for a locking request. If not supported, the Hibernate dialect ignores this query hint.

> The `javax.persistence.lock.scope` is not yet supported (https://hibernate.atlassian.net/browse/HHH-9636) as specified by the JPA standard.

10.7. The `buildLockRequest` API

Traditionally, Hibernate offered the `Session#lock()` method for acquiring an optimistic or a pessimistic lock on a given entity. Because varying the locking options was difficult when using a single `LockMode` parameter, Hibernate has added the `Session#buildLockRequest()` method API.

The following example shows how to obtain shared database lock without waiting for the lock acquisition request.

Example 303. `buildLockRequest` *example*

JAVA

```
Person person = entityManager.find( Person.class, id );
Session session = entityManager.unwrap( Session.class );
session
    .buildLockRequest( LockOptions.NONE )
    .setLockMode( LockMode.PESSIMISTIC_READ )
    .setTimeOut( LockOptions.NO_WAIT )
    .lock( person );
```

```
SQL
SELECT p.id AS id1_0_0_ ,
       p.name AS name2_0_0_
FROM   Person p
WHERE  p.id = 1

SELECT id
FROM   Person
WHERE  id = 1
FOR    SHARE NOWAIT
```

10.8. Follow-on-locking

When using Oracle, the `FOR UPDATE` exclusive locking clause (https://docs.oracle.com/database/121/SQLRF/statements_10002.htm#SQLRF55371) cannot be used with:

- `DISTINCT`
- `GROUP BY`
- `UNION`
- inlined views (derived tables), therefore, affecting the legacy Oracle pagination mechanism as well.

For this reason, Hibernate uses secondary selects to lock the previously fetched entities.

Example 304. Follow-on-locking example

```
JAVA
List<Person> persons = entityManager.createQuery(
    "select DISTINCT p from Person p", Person.class)
.setLockMode( LockModeType.PESSIMISTIC_WRITE )
.getResultList();
```

```
SQL
SELECT DISTINCT p.id as id1_0_, p."name" as name2_0_
FROM Person p

SELECT id
FROM Person
WHERE id = 1 FOR UPDATE

SELECT id
FROM Person
WHERE id = 1 FOR UPDATE
```

> To avoid the N+1 query problem, a separate query can be used to apply the lock using the associated entity identifiers.

Example 305. Secondary query entity locking

JAVA

```
List<Person> persons = entityManager.createQuery(
    "select DISTINCT p from Person p", Person.class)
.getResultList();

entityManager.createQuery(
    "select p.id from Person p where p in :persons")
.setLockMode( LockModeType.PESSIMISTIC_WRITE )
.setParameter( "persons", persons )
.getResultList();
```

SQL

```
SELECT DISTINCT p.id as id1_0_, p."name" as name2_0_
FROM Person p

SELECT p.id as col_0_0_
FROM Person p
WHERE p.id IN ( 1 , 2 )
FOR UPDATE
```

The lock request was moved from the original query to a secondary one which takes the previously fetched entities to lock their associated database records.

Prior to Hibernate 5.2.1, the the follow-on-locking mechanism was applied uniformly to any locking query executing on Oracle. Since 5.2.1, the Oracle Dialect tries to figure out if the current query demand the follow-on-locking mechanism.

Even more important is that you can overrule the default follow-on-locking detection logic and explicitly enable or disable it on a per query basis.

Example 306. Disabling the follow-on-locking mechanism explicitly

```
JAVA
List<Person> persons = entityManager.createQuery(
    "select p from Person p", Person.class)
.setMaxResults( 10 )
.unwrap( Query.class )
.setLockOptions(
    new LockOptions( LockMode.PESSIMISTIC_WRITE )
        .setFollowOnLocking( false ) )
.getResultList();
```

```
SQL
SELECT *
FROM (
    SELECT p.id as id1_0_, p."name" as name2_0_
    FROM Person p
)
WHERE rownum <= 10
FOR UPDATE
```

> The follow-on-locking mechanism should be explicitly enabled only if the current executing query fails because the `FOR UPDATE` clause cannot be applied, meaning that the Dialect resolving mechanism needs to be further improved.

11. Fetching

Fetching, essentially, is the process of grabbing data from the database and making it available to the application. Tuning how an application does fetching is one of the biggest factors in determining how an application will perform. Fetching too much data, in terms of width (values/columns) and/or depth (results/rows), adds unnecessary overhead in terms of both JDBC communication and ResultSet processing. Fetching too little data might cause additional fetching to be needed. Tuning how an application fetches data presents a great opportunity to influence the application overall performance.

11.1. The basics

The concept of fetching breaks down into two different questions.

- When should the data be fetched? Now? Later?
- How should the data be fetched?

> "now" is generally termed eager or immediate. "later" is generally termed lazy or delayed.

There are a number of scopes for defining fetching:

static

Static definition of fetching strategies is done in the mappings. The statically-defined fetch strategies is used in the absence of any dynamically defined strategies

SELECT

Performs a separate SQL select to load the data. This can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed). This is the strategy generally termed N+1.

JOIN

Inherently an EAGER style of fetching. The data to be fetched is obtained through the use of an SQL outer join.

BATCH

Performs a separate SQL select to load a number of related data items using an IN-restriction as part of the SQL WHERE-clause based on a batch size. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).

SUBSELECT

Performs a separate SQL select to load associated data based on the SQL restriction used to load the owner. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).

dynamic (sometimes referred to as runtime)

Dynamic definition is really use-case centric. There are multiple ways to define dynamic fetching:

fetch profiles

defined in mappings, but can be enabled/disabled on the `Session`.

HQL/JPQL

and both Hibernate and JPA Criteria queries have the ability to specify fetching, specific to said query.

entity graphs

Starting in Hibernate 4.2 (JPA 2.1) this is also an option.

11.2. Direct fetching vs entity queries

To see the difference between direct fetching and entity queries in regard to eagerly fetched associations, consider the following entities:

Example 307. Domain model

JAVA

```
@Entity(name = "Department")
public static class Department {

    @Id
    private Long id;

    //Getters and setters omitted for brevity
}

@Entity(name = "Employee")
public static class Employee {

    @Id
    private Long id;

    @NaturalId
    private String username;

    @ManyToOne(fetch = FetchType.EAGER)
    private Department department;

    //Getters and setters omitted for brevity
}
```

The `Employee` entity has a `@ManyToOne` association to a `Department` which is fetched eagerly.

When issuing a direct entity fetch, Hibernate executed the following SQL query:

Example 308. Direct fetching example

JAVA

```
Employee employee = entityManager.find( Employee.class, 1L );
```

SQL

```
select
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.username as username2_1_0_,
    d.id as id1_0_1_
from
    Employee e
left outer join
    Department d
        on e.department_id=d.id
where
    e.id = 1
```

The `LEFT JOIN` clause is added to the generated SQL query because this association is required to be fetched eagerly.

On the other hand, if you are using an entity query that does not contain a `JOIN FETCH` directive to the `Department` association:

Example 309. Entity query fetching example

JAVA

```
Employee employee = entityManager.createQuery(
        "select e " +
        "from Employee e " +
        "where e.id = :id", Employee.class)
.setParameter( "id", 1L )
.getSingleResult();
```

SQL

```
select
    e.id as id1_1_,
    e.department_id as departme3_1_,
    e.username as username2_1_
from
    Employee e
where
    e.id = 1

select
    d.id as id1_0_0_
from
    Department d
where
    d.id = 1
```

Hibernate uses a secondary select instead. This is because the entity query fetch policy cannot be overridden, so Hibernate requires a secondary select to ensure that the EAGER association is fetched prior to returning the result to the user.

> If you forget to JOIN FETCH all EAGER associations, Hibernate is going to issue a secondary select for each and every one of those which, in turn, can lean to N+1 query issues.
>
> For this reason, you should prefer LAZY associations.

11.3. Applying fetch strategies

Let's consider these topics as it relates to an simple domain model and a few use cases.

Example 310. Sample domain model

JAVA

```java
@Entity(name = "Department")
public static class Department {

    @Id
    private Long id;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees = new ArrayList<>();

    //Getters and setters omitted for brevity
}

@Entity(name = "Employee")
public static class Employee {

    @Id
    private Long id;

    @NaturalId
    private String username;

    @Column(name = "pswd")
    @ColumnTransformer(
        read = "decrypt( 'AES', '00', pswd  )",
        write = "encrypt('AES', '00', ?)"
    )
    private String password;

    private int accessLevel;

    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;

    @ManyToMany(mappedBy = "employees")
    private List<Project> projects = new ArrayList<>();

    //Getters and setters omitted for brevity
}

@Entity(name = "Project")
public class Project {

    @Id
    private Long id;

    @ManyToMany
    private List<Employee> employees = new ArrayList<>();

    //Getters and setters omitted for brevity
}
```

> The Hibernate recommendation is to statically mark all associations lazy and to use dynamic fetching strategies for eagerness. This is unfortunately at odds with the JPA specification which defines that all one-to-one and many-to-one associations should be eagerly fetched by default. Hibernate, as a JPA provider, honors that default.

11.4. No fetching

For the first use case, consider the application login process for an `Employee`. Let's assume that login only requires access to the `Employee` information, not `Project` nor `Department` information.

Example 311. No fetching example

JAVA

```
Employee employee = entityManager.createQuery(
    "select e " +
    "from Employee e " +
    "where " +
    "    e.username = :username and " +
    "    e.password = :password",
    Employee.class)
.setParameter( "username", username)
.setParameter( "password", password)
.getSingleResult();
```

In this example, the application gets the `Employee` data. However, because all associations from `Employee` are declared as LAZY (JPA defines the default for collections as LAZY) no other data is fetched.

If the login process does not need access to the `Employee` information specifically, another fetching optimization here would be to limit the width of the query results.

Example 312. No fetching (scalar) example

```
Integer accessLevel = entityManager.createQuery(
    "select e.accessLevel " +
    "from Employee e " +
    "where " +
    "    e.username = :username and " +
    "    e.password = :password",
    Integer.class)
.setParameter( "username", username)
.setParameter( "password", password)
.getSingleResult();
```

JAVA

11.5. Dynamic fetching via queries

For the second use case, consider a screen displaying the `Projects` for an `Employee`. Certainly access to the `Employee` is needed, as is the collection of `Projects` for that Employee. Information about `Departments`, other `Employees` or other `Projects` is not needed.

Example 313. Dynamic JPQL fetching example

```
Employee employee = entityManager.createQuery(
    "select e " +
    "from Employee e " +
    "left join fetch e.projects " +
    "where " +
    "    e.username = :username and " +
    "    e.password = :password",
    Employee.class)
.setParameter( "username", username)
.setParameter( "password", password)
.getSingleResult();
```

JAVA

Example 314. Dynamic query fetching example

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = builder.createQuery( Employee.class );
Root<Employee> root = query.from( Employee.class );
root.fetch( "projects", JoinType.LEFT);
query.select(root).where(
    builder.and(
        builder.equal(root.get("username"), username),
        builder.equal(root.get("password"), password)
    )
);
Employee employee = entityManager.createQuery( query ).getSingleResult();
```

JAVA

In this example we have an `Employee` and their `Projects` loaded in a single query shown both as an HQL query and a JPA Criteria query. In both cases, this resolves to exactly one database query to get all that information.

11.6. Dynamic fetching via JPA entity graph

JPA 2.1 introduced entity graphs so the application developer has more control over fetch plans.

Example 315. Fetch graph example

JAVA

```
@Entity(name = "Employee")
@NamedEntityGraph(name = "employee.projects",
    attributeNodes = @NamedAttributeNode("projects")
)
```

JAVA

```
Employee employee = entityManager.find(
    Employee.class,
    userId,
    Collections.singletonMap(
        "javax.persistence.fetchgraph",
        entityManager.getEntityGraph( "employee.projects" )
    )
);
```

> Entity graphs are the way to override the EAGER fetching associations at runtime. With JPQL, if an EAGER association is omitted, Hibernate will issue a secondary select for every association needed to be fetched eagerly.

11.7. Dynamic fetching via Hibernate profiles

Suppose we wanted to leverage loading by natural-id to obtain the `Employee` information in the "projects for and employee" use-case. Loading by natural-id uses the statically defined fetching strategies, but does not expose a means to define load-specific fetching. So we would leverage a fetch profile.

Example 316. Fetch profile example

```
@Entity(name = "Employee")
@FetchProfile(
    name = "employee.projects",
    fetchOverrides = {
        @FetchProfile.FetchOverride(
            entity = Employee.class,
            association = "projects",
            mode = FetchMode.JOIN
        )
    }
)
```

JAVA

```
session.enableFetchProfile( "employee.projects" );
Employee employee = session.bySimpleNaturalId( Employee.class ).load( username );
```

JAVA

Here the `Employee` is obtained by natural-id lookup and the Employee's `Project` data is fetched eagerly. If the `Employee` data is resolved from cache, the `Project` data is resolved on its own. However, if the `Employee` data is not resolved in cache, the `Employee` and `Project` data is resolved in one SQL query via join as we saw above.

11.8. Batch fetching

Hibernate offers the `@BatchSize` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/BatchSize.html) annotation, which can be used when fetching uninitialized entity proxies.

Considering the following entity mapping:

Example 317. `@BatchSize` *mapping example*

```java
@Entity(name = "Department")
public static class Department {

    @Id
    private Long id;

    @OneToMany(mappedBy = "department")
    //@BatchSize(size = 5)
    private List<Employee> employees = new ArrayList<>();

    //Getters and setters omitted for brevity

}

@Entity(name = "Employee")
public static class Employee {

    @Id
    private Long id;

    @NaturalId
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;

    //Getters and setters omitted for brevity
}
```

Considering that we have previously fetched several `Department` entities, and now we need to initialize the `employees` entity collection for each particular `Department`, the `@BatchSize` annotations allows us to load multiple `Employee` entities in a single database roundtrip.

Example 318. `@BatchSize` *fetching example*

```java
List<Department> departments = entityManager.createQuery(
    "select d " +
    "from Department d " +
    "inner join d.employees e " +
    "where e.name like 'John%'", Department.class)
.getResultList();

for ( Department department : departments ) {
    log.infof(
        "Department %d has {} employees",
        department.getId(),
        department.getEmployees().size()
    );
}
```

```sql
SELECT
    d.id as id1_0_
FROM
    Department d
INNER JOIN
    Employee employees1_
    ON d.id=employees1_.department_id

SELECT
    e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.name as name2_1_0_
FROM
    Employee e
WHERE
    e.department_id IN (
        0, 2, 3, 4, 5
    )

SELECT
    e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.name as name2_1_0_
FROM
    Employee e
WHERE
    e.department_id IN (
        6, 7, 8, 9, 1
    )
```

As you can see in the example above, there are only two SQL statements used to fetch the `Employee` entities associated to multiple `Department` entities.

> Without `@BatchSize`, you'd run into a N+1 query issue, so, instead of 2 SQL statements, there would be 10 queries needed for fetching the `Employee` child entities.
>
> However, although `@BatchSize` is better than running into an N+1 query issue, most of the time, a DTO projection or a `JOIN FETCH` is a much better alternative since it allows you to fetch all the required data with a single query.

11.9. The `@Fetch` annotation mapping

Besides the `FetchType.LAZY` or `FetchType.EAGER` JPA annotations, you can also use the Hibernate-specific `@Fetch` annotation that accepts one of the following `FetchMode`s:

SELECT

Use a secondary select for each individual entity, collection, or join load.

JOIN

Use an outer join to load the related entities, collections or joins.

SUBSELECT

Available for collections only. When accessing a non-initialized collection, this fetch mode will trigger loading all elements of all collections of the same role for all owners associated with the persistence context using a single secondary select.

11.10. `FetchMode.SELECT`

To demonstrate how `FetchMode.SELECT` works, consider the following entity mapping:

Example 319. `FetchMode.SELECT` *mapping example*

```java
@Entity(name = "Department")
public static class Department {

    @Id
    private Long id;

    @OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
    @Fetch(FetchMode.SELECT)
    private List<Employee> employees = new ArrayList<>();

    //Getters and setters omitted for brevity

}

@Entity(name = "Employee")
public static class Employee {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String username;

    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;

    //Getters and setters omitted for brevity

}
```

Considering there are multiple `Department` entities, each one having multiple `Employee` entities, when executing the following test case, Hibernate fetches every uninitialized `Employee` collection using a secondary `SELECT` statement upon accessing the child collection for the first time:

Example 320. `FetchMode.SELECT` *mapping example*

```java
List<Department> departments = entityManager.createQuery(
    "select d from Department d", Department.class )
.getResultList();

log.infof( "Fetched %d Departments", departments.size());

for (Department department : departments ) {
    assertEquals( 3, department.getEmployees().size() );
}
```

```
SQL
SELECT
    d.id as id1_0_
FROM
    Department d

-- Fetched 2 Departments

SELECT
    e.department_id as departme3_1_0_,
    e.id as id1_1_0_,
    e.id as id1_1_1_,
    e.department_id as departme3_1_1_,
    e.username as username2_1_1_
FROM
    Employee e
WHERE
    e.department_id = 1

SELECT
    e.department_id as departme3_1_0_,
    e.id as id1_1_0_,
    e.id as id1_1_1_,
    e.department_id as departme3_1_1_,
    e.username as username2_1_1_
FROM
    Employee e
WHERE
    e.department_id = 2
```

The more `Department` entities are fetched by the first query, the more secondary `SELECT` statements are executed to initialize the `employees` collections. Therefore, `FetchMode.SELECT` can lead to N+1 query issues.

11.11. `FetchMode.SUBSELECT`

To demonstrate how `FetchMode.SUBSELECT` works, we are going to modify the `FetchMode.SELECT` mapping example to use `FetchMode.SUBSELECT`:

Example 321. `FetchMode.SUBSELECT` *mapping example*

```
JAVA
@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
@Fetch(FetchMode.SUBSELECT)
private List<Employee> employees = new ArrayList<>();
```

Now, we are going to fetch all `Department` entities that match a given filtering criteria and then navigate their `employees` collections.

Hibernate is going to avoid the N+1 query issue by generating a single SQL statement to initialize all `employees` collections for all `Department` entities that were previously fetched. Instead of using passing all entity identifiers, Hibernate simply reruns the previous query that fetched the `Department` entities.

Example 322. `FetchMode.SUBSELECT` *mapping example*

JAVA

```
List<Department> departments = entityManager.createQuery(
    "select d " +
    "from Department d " +
    "where d.name like :token", Department.class )
.setParameter( "token", "Department%" )
.getResultList();

log.infof( "Fetched %d Departments", departments.size());

for (Department department : departments ) {
    assertEquals( 3, department.getEmployees().size() );
}
```

SQL

```
SELECT
    d.id as id1_0_
FROM
    Department d
where
    d.name like 'Department%'

-- Fetched 2 Departments

SELECT
    e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.username as username2_1_0_
FROM
    Employee e
WHERE
    e.department_id in (
        SELECT
            fetchmodes0_.id
        FROM
            Department fetchmodes0_
        WHERE
            d.name like 'Department%'
    )
```

11.12. `FetchMode.JOIN`

To demonstrate how `FetchMode.JOIN` works, we are going to modify the `FetchMode.SELECT` mapping example to use `FetchMode.JOIN` instead:

Example 323. `FetchMode.JOIN` *mapping example*

```java
@OneToMany(mappedBy = "department")
@Fetch(FetchMode.JOIN)
private List<Employee> employees = new ArrayList<>();
```

Now, we are going to fetch one `Department` and navigate its `employees` collections.

> The reason why we are not using a JPQL query to fetch multiple `Department` entities is because the `FetchMode.JOIN` strategy would be overridden by the query fetching directive.
>
> To fetch multiple relationships with a JPQL query, the `JOIN FETCH` directive must be used instead.

Therefore, `FetchMode.JOIN` is useful for when entities are fetched directly, via their identifier or natural-id.

Also, the `FetchMode.JOIN` acts as a `FetchType.EAGER` strategy. Even if we mark the association as `FetchType.LAZY`, the `FetchMode.JOIN` will load the association eagerly.

Hibernate is going to avoid the secondary query by issuing an OUTER JOIN for the `employees` collection.

Example 324. `FetchMode.JOIN` *mapping example*

```java
Department department = entityManager.find( Department.class, 1L );

log.infof( "Fetched department: %s", department.getId());

assertEquals( 3, department.getEmployees().size() );
```

```sql
SELECT
    d.id as id1_0_0_,
    e.department_id as departme3_1_1_,
    e.id as id1_1_1_,
    e.id as id1_1_2_,
    e.department_id as departme3_1_2_,
    e.username as username2_1_2_
FROM
    Department d
LEFT OUTER JOIN
    Employee e
        on d.id = e.department_id
WHERE
    d.id = 1

-- Fetched department: 1
```

This time, there was no secondary query because the child collection was loaded along with the parent entity.

11.13. `@LazyCollection`

The `@LazyCollection` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyCollection.html) annotation is used to specify the lazy fetching behavior of a given collection. The possible values are given by the `LazyCollectionOption` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyCollectionOption.html) enumeration:

`TRUE`
: Load it when the state is requested.

`FALSE`
: Eagerly load it.

`EXTRA`
: Prefer extra queries over full collection loading.

The `TRUE` and `FALSE` values are deprecated since you should be using the JPA `FetchType` (http://docs.oracle.com/javaee/7/api/javax/persistence/FetchType.html) attribute of the `@ElementCollection`, `@OneToMany`, or `@ManyToMany` collection.

The `EXTRA` value has no equivalent in the JPA specification, and it's used to avoid loading the entire collection even when the collection is accessed for the first time. Each element is fetched individually using a secondary query.

Example 325. `LazyCollectionOption.EXTRA` *mapping example*

```java
@Entity(name = "Department")
public static class Department {

    @Id
    private Long id;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    @OrderColumn(name = "order_id")
    @LazyCollection( LazyCollectionOption.EXTRA )
    private List<Employee> employees = new ArrayList<>();

    //Getters and setters omitted for brevity

}

@Entity(name = "Employee")
public static class Employee {

    @Id
    private Long id;

    @NaturalId
    private String username;

    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;

    //Getters and setters omitted for brevity

}
```

> `LazyCollectionOption.EXTRA` only works for ordered collections, either List(s) that are annotated with @OrderColumn or Map(s).
>
> For bags (e.g. regular List(s) of entities that do not preserve any certain ordering), the @LazyCollection(LazyCollectionOption.EXTRA)` behaves like any other `FetchType.LAZY` collection (the collection is fetched entirely upon being accessed for the first time).

Now, considering we have the following entities:

Example 326. `LazyCollectionOption.EXTRA` *Domain Model example*

```java
Department department = new Department();
department.setId( 1L );
entityManager.persist( department );

for (long i = 1; i <= 3; i++ ) {
    Employee employee = new Employee();
    employee.setId( i );
    employee.setUsername( String.format( "user_%d", i ) );
    department.addEmployee(employee);
}
```

When fetching the `employee` collection entries by their position in the `List`, Hibernate generates the following SQL statements:

Example 327. `LazyCollectionOption.EXTRA` *fetching example*

```java
Department department = entityManager.find(Department.class, 1L);

int employeeCount = department.getEmployees().size();

for(int i = 0; i < employeeCount; i++ ) {
    log.infof( "Fetched employee: %s", department.getEmployees().get( i ).getUsername());
}
```

SQL

```
SELECT
    max(order_id) + 1
FROM
    Employee
WHERE
    department_id = ?

-- binding parameter [1] as [BIGINT] - [1]

SELECT
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.username as username2_1_0_
FROM
    Employee e
WHERE
    e.department_id=?
    AND e.order_id=?

-- binding parameter [1] as [BIGINT]  - [1]
-- binding parameter [2] as [INTEGER] - [0]

SELECT
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.username as username2_1_0_
FROM
    Employee e
WHERE
    e.department_id=?
    AND e.order_id=?

-- binding parameter [1] as [BIGINT]  - [1]
-- binding parameter [2] as [INTEGER] - [1]

SELECT
    e.id as id1_1_0_,
    e.department_id as departme3_1_0_,
    e.username as username2_1_0_
FROM
    Employee e
WHERE
    e.department_id=?
    AND e.order_id=?

-- binding parameter [1] as [BIGINT]  - [1]
-- binding parameter [2] as [INTEGER] - [2]
```

> Therefore, the child entities were fetched one after the other without triggering a full collection initialization.
>
> For this reason, caution is advised because `LazyCollectionOption.EXTRA` lazy collections are prone to N+1 query issues.

12. Batching

12.1. JDBC batching

JDBC offers support for batching together SQL statements that can be represented as a single PreparedStatement. Implementation wise this generally means that drivers will send the batched operation to the server in one call, which can save on network calls to the database. Hibernate can leverage JDBC batching. The following settings control this behavior.

`hibernate.jdbc.batch_size`

Controls the maximum number of statements Hibernate will batch together before asking the driver to execute the batch. Zero or a negative number disables this feature.

`hibernate.jdbc.batch_versioned_data`

Some JDBC drivers return incorrect row counts when a batch is executed. If your JDBC driver falls into this category this setting should be set to `false`. Otherwise, it is safe to enable this which will allow Hibernate to still batch the DML for versioned entities and still use the returned row counts for optimistic lock checks. Since 5.0, it defaults to true. Previously (versions 3.x and 4.x), it used to be false.

`hibernate.jdbc.batch.builder`

Names the implementation class used to manage batching capabilities. It is almost never a good idea to switch from Hibernate's default implementation. But if you wish to, this setting would name the `org.hibernate.engine.jdbc.batch.spi.BatchBuilder` implementation to use.

`hibernate.order_updates`

Forces Hibernate to order SQL updates by the entity type and the primary key value of the items being updated. This allows for more batching to be used. It will also result in fewer transaction deadlocks in highly concurrent systems. Comes with a performance hit, so benchmark before and after to see if this actually helps or hurts your application.

`hibernate.order_inserts`

Forces Hibernate to order inserts to allow for more batching to be used. Comes with a performance hit, so benchmark before and after to see if this actually helps or hurts your application.

> Since version 5.2, Hibernate allows overriding the global JDBC batch size given by the `hibernate.jdbc.batch_size` configuration property for a given `Session`.

Example 328. Hibernate specific JDBC batch size configuration on a per `Session` basis

```
JAVA
entityManager
    .unwrap( Session.class )
    .setJdbcBatchSize( 10 );
```

12.2. Session batching

The following example shows an anti-pattern for batch inserts.

Example 329. Naive way to insert 100 000 entities with Hibernate

```java
EntityManager entityManager = null;
EntityTransaction txn = null;
try {
    entityManager = entityManagerFactory().createEntityManager();

    txn = entityManager.getTransaction();
    txn.begin();

    for ( int i = 0; i < 100_000; i++ ) {
        Person Person = new Person( String.format( "Person %d", i ) );
        entityManager.persist( Person );
    }

    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
        throw e;
} finally {
    if (entityManager != null) {
        entityManager.close();
    }
}
```

There are several problems associated with this example:

1. Hibernate caches all the newly inserted `Customer` instances in the session-level c1ache, so, when the transaction ends, 100 000 entities are managed by the persistence context. If the maximum memory allocated to the JVM is rather low, this example could fails with an `OutOfMemoryException`. The Java 1.8 JVM allocated either 1/4 of available RAM or 1Gb, which can easily accommodate 100 000 objects on the heap.
2. long-running transactions can deplete a connection pool so other transactions don't get a chance to proceed.
3. JDBC batching is not enabled by default, so every insert statement requires a database roundtrip. To enable JDBC batching, set the `hibernate.jdbc.batch_size` property to an integer between 10 and 50.

> Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.

12.2.1. Batch inserts

When you make new objects persistent, employ methods `flush()` and `clear()` to the session regularly, to control the size of the first-level cache.

Example 330. Flushing and clearing the `Session`

JAVA

```
EntityManager entityManager = null;
EntityTransaction txn = null;
try {
    entityManager = entityManagerFactory().createEntityManager();

    txn = entityManager.getTransaction();
    txn.begin();

    int batchSize = 25;

    for ( int i = 0; i < entityCount; ++i ) {
        Person Person = new Person( String.format( "Person %d", i ) );
        entityManager.persist( Person );

        if ( i > 0 && i % batchSize == 0 ) {
            //flush a batch of inserts and release memory
            entityManager.flush();
            entityManager.clear();
        }
    }

    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
        throw e;
} finally {
    if (entityManager != null) {
        entityManager.close();
    }
}
```

12.2.2. Session scroll

When you retrieve and update data, `flush()` and `clear()` the session regularly. In addition, use method `scroll()` to take advantage of server-side cursors for queries that return many rows of data.

Example 331. Using `scroll()`

JAVA

```
EntityManager entityManager = null;
EntityTransaction txn = null;
ScrollableResults scrollableResults = null;
try {
    entityManager = entityManagerFactory().createEntityManager();

    txn = entityManager.getTransaction();
    txn.begin();

    int batchSize = 25;

    Session session = entityManager.unwrap( Session.class );

    scrollableResults = session
        .createQuery( "select p from Person p" )
        .setCacheMode( CacheMode.IGNORE )
        .scroll( ScrollMode.FORWARD_ONLY );

    int count = 0;
    while ( scrollableResults.next() ) {
        Person Person = (Person) scrollableResults.get( 0 );
        processPerson(Person);
        if ( ++count % batchSize == 0 ) {
            //flush a batch of updates and release memory:
            entityManager.flush();
            entityManager.clear();
        }
    }

    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.isActive()) txn.rollback();
        throw e;
} finally {
    if (scrollableResults != null) {
        scrollableResults.close();
    }
    if (entityManager != null) {
        entityManager.close();
    }
}
```

> If left unclosed by the application, Hibernate will automatically close the underlying resources (e.g. `ResultSet` and `PreparedStatement`) used internally by the `ScrollableResults` when the current transaction is ended (either commit or rollback).
>
> However, it is good practice to close the `ScrollableResults` explicitly.

12.2.3. StatelessSession

`StatelessSession` is a command-oriented API provided by Hibernate. Use it to stream data to and from the database in the form of detached objects. A `StatelessSession` has no persistence context associated with it and does not provide many of the higher-level life cycle semantics.

Some of the things not provided by a `StatelessSession` include:

- a first-level cache
- interaction with any second-level or query cache
- transactional write-behind or automatic dirty checking

Limitations of `StatelessSession`:

- Operations performed using a stateless session never cascade to associated instances.
- Collections are ignored by a stateless session.
- Lazy loading of associations is not supported.
- Operations performed via a stateless session bypass Hibernate's event model and interceptors.
- Due to the lack of a first-level cache, Stateless sessions are vulnerable to data aliasing effects.
- A stateless session is a lower-level abstraction that is much closer to the underlying JDBC.

Example 332. Using a `StatelessSession`

```java
StatelessSession statelessSession = null;
Transaction txn = null;
ScrollableResults scrollableResults = null;
try {
    SessionFactory sessionFactory = entityManagerFactory().unwrap( SessionFactory.class );
    statelessSession = sessionFactory.openStatelessSession();

    txn = statelessSession.getTransaction();
    txn.begin();

    scrollableResults = statelessSession
        .createQuery( "select p from Person p" )
        .scroll(ScrollMode.FORWARD_ONLY);

    while ( scrollableResults.next() ) {
        Person Person = (Person) scrollableResults.get( 0 );
        processPerson(Person);
        statelessSession.update( Person );
    }

    txn.commit();
} catch (RuntimeException e) {
    if ( txn != null && txn.getStatus() == TransactionStatus.ACTIVE) txn.rollback();
        throw e;
} finally {
    if (scrollableResults != null) {
        scrollableResults.close();
    }
    if (statelessSession != null) {
        statelessSession.close();
    }
}
```

The `Customer` instances returned by the query are immediately detached. They are never associated with any persistence context.

The `insert()`, `update()`, and `delete()` operations defined by the `StatelessSession` interface operate directly on database rows. They cause the corresponding SQL operations to be executed immediately. They have different semantics from the `save()`, `saveOrUpdate()`, and `delete()` operations defined by the `Session` interface.

12.3. Hibernate Query Language for DML

DML, or Data Manipulation Language, refers to SQL statements such as `INSERT`, `UPDATE`, and `DELETE`. Hibernate provides methods for bulk SQL-style DML statement execution, in the form of Hibernate Query Language (HQL).

12.3.1. HQL/JPQL for UPDATE and DELETE

Both the Hibernate native Query Language and JPQL (Java Persistence Query Language) provide support for bulk UPDATE and DELETE.

Example 333. Psuedo-syntax for UPDATE and DELETE statements using HQL

```
UPDATE FROM EntityName e WHERE e.name = ?

DELETE FROM EntityName e WHERE e.name = ?
```

JAVA

> The `FROM` and `WHERE` clauses are each optional, but it's good practice to use them.

The `FROM` clause can only refer to a single entity, which can be aliased. If the entity name is aliased, any property references must be qualified using that alias. If the entity name is not aliased, then it is illegal for any property references to be qualified.

> Joins, either implicit or explicit, are prohibited in a bulk HQL query. You can use sub-queries in the `WHERE` clause, and the sub-queries themselves can contain joins.

Example 334. Executing a JPQL `UPDATE`, using the `Query.executeUpdate()`

```
int updatedEntities = entityManager.createQuery(
    "update Person p " +
    "set p.name = :newName " +
    "where p.name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();
```

JAVA

Example 335. Executing an HQL `UPDATE`, using the `Query.executeUpdate()`

```java
int updatedEntities = session.createQuery(
    "update Person " +
    "set name = :newName " +
    "where name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();
```

In keeping with the EJB3 specification, HQL `UPDATE` statements, by default, do not effect the version or the timestamp property values for the affected entities. You can use a versioned update to force Hibernate to reset the version or timestamp property values, by adding the `VERSIONED` keyword after the `UPDATE` keyword.

Example 336. Updating the version of timestamp

```java
int updatedEntities = session.createQuery(
    "update versioned Person " +
    "set name = :newName " +
    "where name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();
```

> If you use the `VERSIONED` statement, you cannot use custom version types, which use class `org.hibernate.usertype.UserVersionType`.
>
> This feature is only available in HQL since it's not standardized by JPA.

Example 337. A JPQL `DELETE` statement

```
int deletedEntities = entityManager.createQuery(
    "delete Person p " +
    "where p.name = :name" )
.setParameter( "name", name )
.executeUpdate();
```
JAVA

Example 338. An HQL `DELETE` statement

```
int deletedEntities = session.createQuery(
    "delete Person " +
    "where name = :name" )
.setParameter( "name", name )
.executeUpdate();
```
JAVA

Method `Query.executeUpdate()` returns an `int` value, which indicates the number of entities effected by the operation. This may or may not correlate to the number of rows effected in the database. An JPQL/HQL bulk operation might result in multiple SQL statements being executed, such as for joined-subclass. In the example of joined-subclass, a `DELETE` against one of the subclasses may actually result in deletes in the tables underlying the join, or further down the inheritance hierarchy.

12.3.2. HQL syntax for INSERT

Example 339. Pseudo-syntax for INSERT statements

```
INSERT INTO EntityName
        properties_list
SELECT properties_list
FROM ...
```
JAVA

Only the `INSERT INTO … SELECT …` form is supported. You cannot specify explicit values to insert.

The `properties_list` is analogous to the column specification in the `SQL INSERT` statement. For entities involved in mapped inheritance, you can only use properties directly defined on that given class-level in the `properties_list`. Superclass properties are not allowed and subclass properties are irrelevant. In other words, `INSERT` statements are inherently non-polymorphic.

The SELECT statement can be any valid HQL select query, but the return types must match the types expected by the INSERT. Hibernate verifies the return types during query compilation, instead of expecting the database to check it. Problems might result from Hibernate types which are equivalent, rather than equal. One such example is a mismatch between a property defined as an `org.hibernate.type.DateType` and a property defined as an `org.hibernate.type.TimestampType`, even though the database may not make a distinction, or may be capable of handling the conversion.

If id property is not specified in the `properties_list`, Hibernate generates a value automatically. Automatic generation is only available if you use ID generators which operate on the database. Otherwise, Hibernate throws an exception during parsing. Available in-database generators are `org.hibernate.id.SequenceGenerator` and its subclasses, and objects which implement `org.hibernate.id.PostInsertIdentifierGenerator`. The most notable exception is `org.hibernate.id.TableHiLoGenerator`, which does not expose a selectable way to get its values.

For properties mapped as either version or timestamp, the insert statement gives you two options. You can either specify the property in the properties_list, in which case its value is taken from the corresponding select expressions, or omit it from the properties_list, in which case the seed value defined by the org.hibernate.type.VersionType is used.

Example 340. HQL INSERT statement

JAVA

```
int insertedEntities = session.createQuery(
    "insert into Partner (id, name) " +
    "select p.id, p.name " +
    "from Person p ")
.executeUpdate();
```

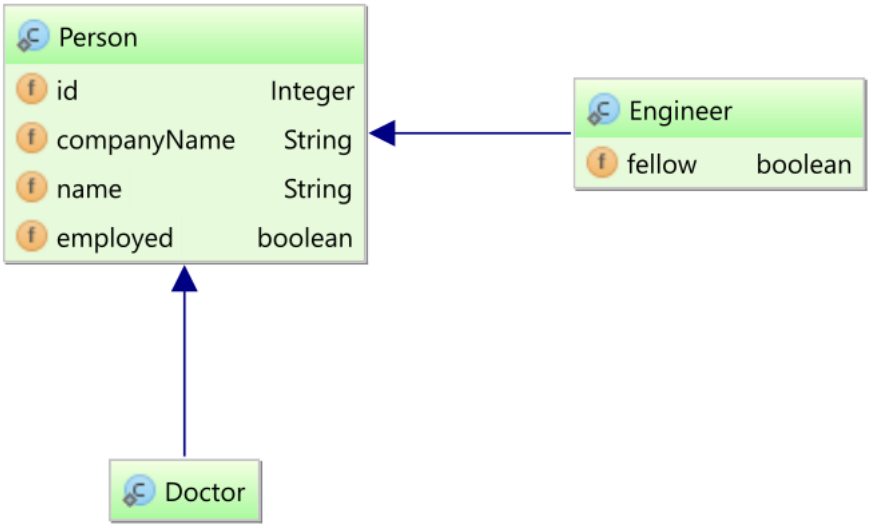
This section is only a brief overview of HQL. For more information, see HQL.

12.3.3. Bulk-id strategies

This article is about the HHH-11262 (https://hibernate.atlassian.net/browse/HHH-11262) JIRA issue which now allows the bulk-id strategies to work even when you cannot create temporary tables.

Class diagram

Considering we have the following entities:



The `Person` entity is the base class of this entity inheritance model, and is mapped as follows:

Example 341. Bulk-id base class entity

JAVA

```
@Entity(name = "Person")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private boolean employed;

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public boolean isEmployed() {
        return employed;
    }

    public void setEmployed(boolean employed) {
        this.employed = employed;
    }
}
```

Both the `Doctor` and `Engineer` entity classes extend the `Person` base class:

Example 342. Bulk-id subclass entities

```java
@Entity(name = "Doctor")
public static class Doctor extends Person {
}

@Entity(name = "Engineer")
public static class Engineer extends Person {

    private boolean fellow;

    public boolean isFellow() {
        return fellow;
    }

    public void setFellow(boolean fellow) {
        this.fellow = fellow;
    }
}
```

Inheritance tree bulk processing

Now, when you try to execute a bulk entity delete query:

Example 343. Bulk-id delete query example

```java
int updateCount = session.createQuery(
    "delete from Person where employed = :employed" )
.setParameter( "employed", false )
.executeUpdate();
```

SQL

```
create temporary table
    HT_Person
(
    id int4 not null,
    companyName varchar(255) not null
)

insert
into
    HT_Person
    select
        p.id as id,
        p.companyName as companyName
    from
        Person p
    where
        p.employed = ?

delete
from
    Engineer
where
    (
        id, companyName
    ) IN (
        select
            id,
            companyName
        from
            HT_Person
    )

delete
from
    Doctor
where
    (
        id, companyName
    ) IN (
        select
            id,
            companyName
        from
            HT_Person
    )

delete
from
    Person
where
    (
        id, companyName
    ) IN (
        select
            id,
            companyName
        from
```

```
            HT_Person
    )
```

`HT_Person` is a temporary table that Hibernate creates to hold all the entity identifiers that are to be updated or deleted by the bulk id operation. The temporary table can be either global or local, depending on the underlying database capabilities.

Non-temporary table bulk-id strategies

As the HHH-11262 (https://hibernate.atlassian.net/browse/HHH-11262) issue describes, there are use cases when the application developer cannot use temporary tables because the database user lacks this privilege.

In this case, we defined several options which you can choose depending on your database capabilities:

- `InlineIdsInClauseBulkIdStrategy`
- `InlineIdsSubSelectValueListBulkIdStrategy`
- `InlineIdsOrClauseBulkIdStrategy`
- `CteValuesListBulkIdStrategy`

`InlineIdsInClauseBulkIdStrategy`

To use this strategy, you need to configure the following configuration property:

XML

```
<property name="hibernate.hql.bulk_id_strategy"
          value="org.hibernate.hql.spi.id.inline.InlineIdsInClauseBulkIdStrategy"
/>
```

Now, when running the previous test case, Hibernate generates the following SQL statements:

Example 344. `InlineIdsInClauseBulkIdStrategy` *delete entity query example*

```SQL
select
    p.id as id,
    p.companyName as companyName
from
    Person p
where
    p.employed = ?

delete
from
    Engineer
where
        ( id, companyName )
    in (
        ( 1,'Red Hat USA' ),
        ( 3,'Red Hat USA' ),
        ( 1,'Red Hat Europe' ),
        ( 3,'Red Hat Europe' )
    )

delete
from
    Doctor
where
        ( id, companyName )
    in (
        ( 1,'Red Hat USA' ),
        ( 3,'Red Hat USA' ),
        ( 1,'Red Hat Europe' ),
        ( 3,'Red Hat Europe' )
    )

delete
from
    Person
where
        ( id, companyName )
    in (
        ( 1,'Red Hat USA' ),
        ( 3,'Red Hat USA' ),
        ( 1,'Red Hat Europe' ),
        ( 3,'Red Hat Europe' )
    )
```

So, the entity identifiers are selected first and used for each particular update or delete statement.

> The IN clause row value expression has long been supported by Oracle, PostgreSQL, and nowadays by MySQL 5.7. However, SQL Server 2014 does not support this syntax, so you'll have to use a different strategy.

`InlineIdsSubSelectValueListBulkIdStrategy`

To use this strategy, you need to configure the following configuration property:

```xml
<property name="hibernate.hql.bulk_id_strategy"
          value="org.hibernate.hql.spi.id.inline.InlineIdsSubSelectValueListBulkIdStrategy"
/>
```

Now, when running the previous test case, Hibernate generates the following SQL statements:

Example 345. `InlineIdsSubSelectValueListBulkIdStrategy` *delete entity query example*

SQL

```
select
    p.id as id,
    p.companyName as companyName
from
    Person p
where
    p.employed = ?

delete
from
    Engineer
where
    ( id, companyName ) in (
        select
            id,
            companyName
        from (
        values
            ( 1,'Red Hat USA' ),
            ( 3,'Red Hat USA' ),
            ( 1,'Red Hat Europe' ),
            ( 3,'Red Hat Europe' )
        ) as HT
            (id, companyName)
    )

delete
from
    Doctor
where
    ( id, companyName ) in (
         select
            id,
            companyName
        from (
        values
            ( 1,'Red Hat USA' ),
            ( 3,'Red Hat USA' ),
            ( 1,'Red Hat Europe' ),
            ( 3,'Red Hat Europe' )
        ) as HT
            (id, companyName)
    )

delete
from
    Person
where
    ( id, companyName ) in (
        select
            id,
            companyName
        from (
        values
            ( 1,'Red Hat USA' ),
            ( 3,'Red Hat USA' ),
```

```
            ( 1,'Red Hat Europe' ),
            ( 3,'Red Hat Europe' )
        ) as HT
            (id, companyName)
    )
```

> The underlying database must support the VALUES list clause, like PostgreSQL or SQL Server 2008. However, this strategy requires the IN-clause row value expression for composite identifiers so you can use this strategy only with PostgreSQL.

`InlineIdsOrClauseBulkIdStrategy`

To use this strategy, you need to configure the following configuration property:

```
<property name="hibernate.hql.bulk_id_strategy"
          value="org.hibernate.hql.spi.id.inline.InlineIdsOrClauseBulkIdStrategy"
/>
```
XML

Now, when running the previous test case, Hibernate generates the following SQL statements:

Example 346. `InlineIdsOrClauseBulkIdStrategy` *delete entity query example*

```
SQL
select
    p.id as id,
    p.companyName as companyName
from
    Person p
where
    p.employed = ?

delete
from
    Engineer
where
    ( id = 1 and companyName = 'Red Hat USA' )
or  ( id = 3 and companyName = 'Red Hat USA' )
or  ( id = 1 and companyName = 'Red Hat Europe' )
or  ( id = 3 and companyName = 'Red Hat Europe' )

delete
from
    Doctor
where
    ( id = 1 and companyName = 'Red Hat USA' )
or  ( id = 3 and companyName = 'Red Hat USA' )
or  ( id = 1 and companyName = 'Red Hat Europe' )
or  ( id = 3 and companyName = 'Red Hat Europe' )

delete
from
    Person
where
    ( id = 1 and companyName = 'Red Hat USA' )
or  ( id = 3 and companyName = 'Red Hat USA' )
or  ( id = 1 and companyName = 'Red Hat Europe' )
or  ( id = 3 and companyName = 'Red Hat Europe' )
```

> This strategy has the advantage of being supported by all the major relational database systems (e.g. Oracle, SQL Server, MySQL, and PostgreSQL).

`CteValuesListBulkIdStrategy`

To use this strategy, you need to configure the following configuration property:

XML

```
<property name="hibernate.hql.bulk_id_strategy"
          value="org.hibernate.hql.spi.id.inline.CteValuesListBulkIdStrategy"
/>
```

Now, when running the previous test case, Hibernate generates the following SQL statements:

Example 347. `CteValuesListBulkIdStrategy` *delete entity query example*

SQL

```
select
    p.id as id,
    p.companyName as companyName
from
    Person p
where
    p.employed = ?

with HT_Person (id,companyName ) as (
    select id, companyName
    from (
    values
        (?, ?),
        (?, ?),
        (?, ?),
        (?, ?)
    ) as HT (id, companyName) )
delete
from
    Engineer
where
    ( id, companyName ) in (
        select
            id, companyName
        from
            HT_Person
    )

with HT_Person (id,companyName ) as (
    select id, companyName
    from (
    values
        (?, ?),
        (?, ?),
        (?, ?),
        (?, ?)
    ) as HT (id, companyName) )
delete
from
    Doctor
where
    ( id, companyName ) in (
        select
            id, companyName
        from
            HT_Person
    )


with HT_Person (id,companyName ) as (
    select id, companyName
    from (
    values
        (?, ?),
        (?, ?),
        (?, ?),
```

```
        (?, ?)
    ) as HT (id, companyName) )
delete
from
    Person
where
    ( id, companyName ) in (
        select
            id, companyName
        from
            HT_Person
    )
```

> The underlying database must support the CTE (Common Table Expressions) that can be referenced from non-query statements as well, like PostgreSQL since 9.1 or SQL Server since 2005. The underlying database must also support the VALUES list clause, like PostgreSQL or SQL Server 2008.
>
> However, this strategy requires the IN-clause row value expression for composite identifiers, so you can only use this strategy only with PostgreSQL.

If you can use temporary tables, that's probably the best choice. However, if you are not allowed to create temporary tables, you must pick one of these four strategies that works with your underlying database. Before making your mind, you should benchmark which one works best for your current workload. For instance, CTE are optimization fences in PostgreSQL (http://blog.2ndquadrant.com/postgresql-ctes-are-optimization-fences/), so make sure you measure before taking a decision.

If you're using Oracle or MySQL 5.7, you can choose either `InlineIdsOrClauseBulkIdStrategy` or `InlineIdsInClauseBulkIdStrategy`. For older version of MySQL, then you can only use `InlineIdsOrClauseBulkIdStrategy`.

If you're using SQL Server, `InlineIdsOrClauseBulkIdStrategy` is the only option for you.

If you're using PostgreSQL, then you have the luxury of choosing any of these four strategies.

13. Caching

At runtime, Hibernate handles moving data into and out of the second-level cache in response to the operations performed by the `Session`, which acts as a transaction-level cache of persistent data. Once an entity becomes managed, that object is added to the internal cache of the current persistence context (`EntityManager` or `Session`). The persistence context is also called the first-level cache, and it's enabled by default.

It is possible to configure a JVM-level (`SessionFactory`-level) or even a cluster cache on a class-by-class and collection-by-collection basis.

> Be aware that caches are not aware of changes made to the persistent store by other applications. They can, however, be configured to regularly expire cached data.

13.1. Configuring second-level caching

Hibernate can integrate with various caching providers for the purpose of caching data outside the context of a particular `Session`. This section defines the settings which control this behavior.

13.1.1. RegionFactory

`org.hibernate.cache.spi.RegionFactory` defines the integration between Hibernate and a pluggable caching provider. `hibernate.cache.region.factory_class` is used to declare the provider to use. Hibernate comes with built-in support for the Java caching standard JCache and also two popular caching libraries: Ehcache and Infinispan. Detailed information is provided later in this chapter.

13.1.2. Caching configuration properties

Besides specific provider configuration, there are a number of configurations options on the Hibernate side of the integration that control various caching behaviors:

`hibernate.cache.use_second_level_cache`

Enable or disable second level caching overall. Default is true, although the default region factory is `NoCachingRegionFactory`.

`hibernate.cache.use_query_cache`

Enable or disable second level caching of query results. Default is false.

`hibernate.cache.query_cache_factory`

Query result caching is handled by a special contract that deals with staleness-based invalidation of the results. The default implementation does not allow stale results at all. Use this for applications that would like to relax that. Names an implementation of `org.hibernate.cache.spi.QueryCacheFactory`

`hibernate.cache.use_minimal_puts`

Optimizes second-level cache operations to minimize writes, at the cost of more frequent reads. Providers typically set this appropriately.

`hibernate.cache.region_prefix`

Defines a name to be used as a prefix to all second-level cache region names.

`hibernate.cache.default_cache_concurrency_strategy`

In Hibernate second-level caching, all regions can be configured differently including the concurrency strategy to use when accessing that particular region. This setting allows to define a default strategy to be used. This setting is very rarely required as the pluggable providers do specify the default strategy to use. Valid values include:

- read-only,
- read-write,
- nonstrict-read-write,
- transactional

`hibernate.cache.use_structured_entries`

If `true` , forces Hibernate to store data in the second-level cache in a more human-friendly format. Can be useful if you'd like to be able to "browse" the data directly in your cache, but does have a performance impact.

`hibernate.cache.auto_evict_collection_cache`

Enables or disables the automatic eviction of a bidirectional association's collection cache entry when the association is changed just from the owning side. This is disabled by default, as it has a performance impact to track this state. However if your application does not manage both sides of bidirectional association where the collection side is cached, the alternative is to have stale data in that collection cache.

`hibernate.cache.use_reference_entries`

Enable direct storage of entity references into the second level cache for read-only or immutable entities.

`hibernate.cache.keys_factory`

When storing entries into second-level cache as key-value pair, the identifiers can be wrapped into tuples <entity type, tenant, identifier> to guarantee uniqueness in case that second-level cache stores all entities in single space. These tuples are then used as keys in the cache. When the second-level cache implementation (incl. its configuration) guarantees that different entity types

are stored separately and multi-tenancy is not used, you can omit this wrapping to achieve better performance. Currently, this property is only supported when Infinispan is configured as the second-level cache implementation. Valid values are:

- `default` (wraps identitifers in the tuple)
- `simple` (uses identifiers as keys without any wrapping)
- fully qualified class name that implements `org.hibernate.cache.spi.CacheKeysFactory`

13.2. Configuring second-level cache mappings

The cache mappings can be configured via JPA annotations or XML descriptors or using the Hibernate-specific mapping files.

By default, entities are not part of the second level cache and we recommend you to stick to this setting. However, 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 in your configuration file. The following values are possible:

`ENABLE_SELECTIVE` (Default and recommended value)

Entities are not cached unless explicitly marked as cacheable (with the `@Cacheable` (https://docs.oracle.com/javaee/7/api/javax/persistence/Cacheable.html) annotation).

`DISABLE_SELECTIVE`

Entities are cached unless explicitly marked as non-cacheable.

`ALL`

Entities are always cached even if marked as non-cacheable.

`NONE`

No entity is 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 globally via the `hibernate.cache.default_cache_concurrency_strategy` configuration property. The values for this property are:

read-only

If your application needs to read, but not modify, instances of a persistent class, a read-only cache is the best choice. Application can still delete entities and these changes should be reflected in second-level cache so that the cache does not provide stale entities. Implementations may use performance optimizations based on the immutability of entities.

read-write

If the application needs to update data, a read-write cache might be appropriate. This strategy provides consistent access to single entity, but not a serializable transaction isolation level; e.g. when TX1 reads looks up an entity and does not find it, TX2 inserts the entity into cache and TX1 looks it up again, the new entity can be read in TX1.

nonstrict-read-write

Similar to read-write strategy but there might be occasional stale reads upon concurrent access to an entity. The choice of this strategy might be appropriate if the application rarely updates the same data simultaneously and strict transaction isolation is not required. Implementations may use performance optimizations that make use of the relaxed consistency guarantee.

transactional

Provides serializable transaction isolation level.

> Rather than using a global cache concurrency strategy, it is recommended to define this setting on a per entity basis. Use the `@org.hibernate.annotations.Cache` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Cache.html) annotation for that.

The `@Cache` annotation define three attributes:

usage

Defines the `CacheConcurrencyStrategy`

region

Defines a cache region where entries will be stored

include

If lazy properties should be included in the second level cache. The default value is `all` so lazy properties are cacheable. The other possible value is `non-lazy` so lazy properties are not cacheable.

13.3. Entity inheritance and second-level cache mapping

When using inheritance, the JPA `@Cacheable` and the Hibernate-specific `@Cache` annotations should be declared at the root-entity level only. That being said, it is not possible to customize the base class `@Cacheable` or `@Cache` definition in subclasses.

Although the JPA 2.1 specification says that the `@Cacheable` annotation could be overwritten by a subclass:

> *The value of the* `Cacheable` *annotation is inherited by subclasses; it can be overridden by specifying* `Cacheable` *on a subclass.*
>
> — *Section 11.1.7 of the JPA 2.1 Specification*

Hibernate requires that a given entity hierarchy share the same caching semantics.

The reasons why Hibernate requires that all entities belonging to an inheritance tree share the same caching definition can be summed as follows:

- from a performance perspective, adding an additional check on a per entity type level would slow the bootstrap process.
- providing different caching semantics for subclasses would violate the Liskov substitution principle (https://en.wikipedia.org/wiki/Liskov_substitution_principle).

  Assuming we have a base class, `Payment` and a subclass `CreditCardPayment`. If the `Payment` is not cacheable and the `CreditCardPayment` is cached, what should happen when executing the following code snippet:

  ```
  JAVA
  Payment payment = entityManager.find(Payment.class, creditCardPaymentId);
  CreditCardPayment creditCardPayment = (CreditCardPayment) CreditCardPayment;
  ```

  In this particular case, the second level cache key is formed of the entity class name and the identifier:

  ```
  JAVA
  keyToLoad = {org.hibernate.engine.spi.EntityKey@4712}
   identifier = {java.lang.Long@4716} "2"
   persister = {org.hibernate.persister.entity.SingleTableEntityPersister@4629}
    "SingleTableEntityPersister(org.hibernate.userguide.model.Payment)"
  ```

  Should Hibernate load the `CreditCardPayment` from the cache as indicated by the actual entity type, or it should not use the cache since the `Payment` is not supposed to be cached?

> Because of all these intricacies, Hibernate only considers the base class `@Cacheable` and `@Cache` definition.

13.4. Entity cache

Example 348. Entity cache mapping

```java
@Entity(name = "Phone")
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    private String mobile;

    @ManyToOne
    private Person person;

    @Version
    private int version;

    public Phone() {}

    public Phone(String mobile) {
        this.mobile = mobile;
    }

    public Long getId() {
        return id;
    }

    public String getMobile() {
        return mobile;
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
}
```

Hibernate stores cached entities in a dehydrated form, which is similar to the database representation. Aside from the foreign key column values of the `@ManyToOne` or `@OneToOne` child-side associations, entity relationships are not stored in the cache,

Once an entity is stored in the second-level cache, you can avoid a database hit and load the entity from the cache alone:

Example 349. Loading entity using JPA

```java
Person person = entityManager.find( Person.class, 1L );
```

Example 350. Loading entity using Hibernate native API

```java
Person person = session.get( Person.class, 1L );
```

The Hibernate second-level cache can also load entities by their natural id:

Example 351. Hibernate natural id entity mapping

```java
@Entity(name = "Person")
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public static class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @NaturalId
    @Column(name = "code", unique = true)
    private String code;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

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

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }
}
```

Example 352. Loading entity using Hibernate native natural id API

```java
Person person = session
    .byNaturalId( Person.class )
    .using( "code", "unique-code")
    .load();
```

13.5. Collection cache

Hibernate can also cache collections, and the `@Cache` annotation must be on added to the collection property.

If the collection is made of value types (basic or embeddables mapped with `@ElementCollection`), the collection is stored as such. If the collection contains other entities (`@OneToMany` or `@ManyToMany`), the collection cache entry will store the entity identifiers only.

Example 353. Collection cache mapping

```java
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
private List<Phone> phones = new ArrayList<>(  );
```

Collections are read-through, meaning they are cached upon being accessed for the first time:

Example 354. Collection cache usage

```java
Person person = entityManager.find( Person.class, 1L );
person.getPhones().size();
```

Subsequent collection retrievals will use the cache instead of going to the database.

> The collection cache is not write-through so any modification will trigger a collection cache entry invalidation. On a subsequent access, the collection will be loaded from the database and re-cached.

13.6. Query cache

Aside from caching entities and collections, Hibernate offers a query cache too. This is useful for frequently executed queries with fixed parameter values.

> Caching of query results introduces some overhead in terms of your applications normal transactional processing. For example, if you cache results of a query against `Person`, Hibernate will need to keep track of when those results should be invalidated because changes have been committed against any `Person` entity.
>
> That, coupled with the fact that most applications simply gain no benefit from caching query results, leads Hibernate to disable caching of query results by default.

To use query caching, you will first need to enable it with the following configuration property:

Example 355. Enabling query cache

```xml
<property
    name="hibernate.cache.use_query_cache"
    value="true" />
```

As mentioned above, most queries do not benefit from caching or their results. So by default, individual queries are not cached even after enabling query caching. Each particular query that needs to be cached must be manually set as cacheable. This way, the query looks for existing cache results or adds the query results to the cache when being executed.

Example 356. Caching query using JPA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name = :name", Person.class)
.setParameter( "name", "John Doe")
.setHint( "org.hibernate.cacheable", "true")
.getResultList();
```

JAVA

Example 357. Caching query using Hibernate native API

```
List<Person> persons = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name = :name")
.setParameter( "name", "John Doe")
.setCacheable(true)
.list();
```

JAVA

> The query cache does not cache the state of the actual entities in the cache; it caches only identifier values and results of value type.
>
> Just as with collection caching, the query cache should always be used in conjunction with the second-level cache for those entities expected to be cached as part of a query result cache.

13.6.1. Query cache regions

This setting creates two new cache regions:

`org.hibernate.cache.internal.StandardQueryCache`
Holding the cached query results

`org.hibernate.cache.spi.UpdateTimestampsCache`
Holding timestamps of the most recent updates to queryable tables. These are used to validate the results as they are served from the query cache.

> If you configure your underlying cache implementation to use expiration, it's very important that the timeout of the underlying cache region for the `UpdateTimestampsCache` is set to a higher value than the timeouts of any of the query caches.
>
> In fact, we recommend that the `UpdateTimestampsCache` region is not configured for expiration (time-based) or eviction (size/memory-based) at all. Note that an LRU (Least Recently Used) cache eviction policy is never appropriate for this particular cache region.

If you require fine-grained control over query cache expiration policies, you can specify a named cache region for a particular query.

Example 358. Caching query in custom region using JPA

```
List<Person> persons = entityManager.createQuery(
        "select p " +
        "from Person p " +
        "where p.id > :id", Person.class)
        .setParameter( "id", 0L)
        .setHint( QueryHints.HINT_CACHEABLE, "true")
        .setHint( QueryHints.HINT_CACHE_REGION, "query.cache.person" )
        .getResultList();
```

JAVA

Example 359. Caching query in custom region using Hibernate native API

```
List<Person> persons = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.id > :id")
.setParameter( "id", 0L)
.setCacheable(true)
.setCacheRegion( "query.cache.person" )
.list();
```

JAVA

If you want to force the query cache to refresh one of its regions (disregarding any cached results it finds there), you can use custom cache modes.

Example 360. Using custom query cache mode with JPA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.id > :id", Person.class)
.setParameter( "id", 0L)
.setHint( QueryHints.HINT_CACHEABLE, "true")
.setHint( QueryHints.HINT_CACHE_REGION, "query.cache.person" )
.setHint( "javax.persistence.cache.storeMode", CacheStoreMode.REFRESH )
.getResultList();
```
JAVA

Example 361. Using custom query cache mode with Hibernate native API

```
List<Person> persons = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.id > :id")
.setParameter( "id", 0L)
.setCacheable(true)
.setCacheRegion( "query.cache.person" )
.setCacheMode( CacheMode.REFRESH )
.list();
```
JAVA

When using `CacheStoreMode.REFRESH` (http://docs.oracle.com/javaee/7/api/javax/persistence/CacheStoreMode.html#REFRESH) or `CacheMode.REFRESH` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/CacheMode.html#REFRESH) in conjunction with the region you have defined for the given query, Hibernate will selectively force the results cached in that particular region to be refreshed.

This is particularly useful in cases where underlying data may have been updated via a separate process and is a far more efficient alternative to bulk eviction of the region via `SessionFactory` eviction which looks as follows:

```
session.getSessionFactory().getCache().evictQueryRegion( "query.cache.person" );
```
JAVA

13.7. Managing the cached data

Traditionally, Hibernate defined the `CacheMode` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/CacheMode.html) enumeration to describe the ways of interactions with the cached data. JPA split cache modes by storage (`CacheStoreMode` (http://docs.oracle.com/javaee/7/api/javax/persistence/CacheStoreMode.html)) and retrieval (`CacheRetrieveMode` (http://docs.oracle.com/javaee/7/api/javax/persistence/CacheRetrieveMode.html)).

The relationship between Hibernate and JPA cache modes can be seen in the following table:

Table 5. Cache modes relationships

Hibernate	JPA	Description
`CacheMode.NORMAL`	`CacheStoreMode.USE` and `CacheRetrieveMode.USE`	Default. Reads/writes data from/into cache
`CacheMode.REFRESH`	`CacheStoreMode.REFRESH` and `CacheRetrieveMode.BYPASS`	Doesn't read from cache, but writes to the cache upon loading from the database
`CacheMode.PUT`	`CacheStoreMode.USE` and `CacheRetrieveMode.BYPASS`	Doesn't read from cache, but writes to the cache as it reads from the database
`CacheMode.GET`	`CacheStoreMode.BYPASS` and `CacheRetrieveMode.USE`	Read from the cache, but doesn't write to cache
`CacheMode.IGNORE`	`CacheStoreMode.BYPASS` and `CacheRetrieveMode.BYPASS`	Doesn't read/write data from/into cache

Setting the cache mode can be done either when loading entities directly or when executing a query.

Example 362. Using custom cache modes with JPA

```
Map<String, Object> hints = new HashMap<>(  );
hints.put( "javax.persistence.cache.retrieveMode " , CacheRetrieveMode.USE );
hints.put( "javax.persistence.cache.storeMode" , CacheStoreMode.REFRESH );
Person person = entityManager.find( Person.class, 1L , hints);
```

JAVA

Example 363. Using custom cache modes with Hibernate native API

```
session.setCacheMode( CacheMode.REFRESH );
Person person = session.get( Person.class, 1L );
```

JAVA

The custom cache modes can be set for queries as well:

Example 364. Using custom cache modes for queries with JPA

```
List<Person> persons = entityManager.createQuery(
    "select p from Person p", Person.class)
.setHint( QueryHints.HINT_CACHEABLE, "true")
.setHint( "javax.persistence.cache.retrieveMode " , CacheRetrieveMode.USE )
.setHint( "javax.persistence.cache.storeMode" , CacheStoreMode.REFRESH )
.getResultList();
```

JAVA

Example 365. Using custom cache modes for queries with Hibernate native API

```
List<Person> persons = session.createQuery(
    "select p from Person p" )
.setCacheable( true )
.setCacheMode( CacheMode.REFRESH )
.list();
```

JAVA

13.7.1. Evicting cache entries

Because the second level cache is bound to the `EntityManagerFactory` or the `SessionFactory`, cache eviction must be done through these two interfaces.

JPA only supports entity eviction through the `javax.persistence.Cache` (https://docs.oracle.com/javaee/7/api/javax/persistence/Cache.html) interface:

Example 366. Evicting entities with JPA

```
entityManager.getEntityManagerFactory().getCache().evict( Person.class );
```

JAVA

Hibernate is much more flexible in this regard as it offers fine-grained control over what needs to be evicted. The `org.hibernate.Cache` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Cache.html) interface defines various evicting strategies:

- entities (by their class or region)
- entities stored using the natural-id (by their class or region)
- collections (by the region, and it might take the collection owner identifier as well)
- queries (by region)

Example 367. Evicting entities with Hibernate native API

```
session.getSessionFactory().getCache().evictQueryRegion( "query.cache.person" );
```
JAVA

13.8. Caching statistics

If you enable the `hibernate.generate_statistics` configuration property, Hibernate will expose a number of metrics via `SessionFactory.getStatistics()`. Hibernate can even be configured to expose these statistics via JMX.

This way, you can get access to the `Statistics` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/stat/Statistics.html) class which comprises all sort of second-level cache metrics.

Example 368. Caching statistics

```
Statistics statistics = session.getSessionFactory().getStatistics();
SecondLevelCacheStatistics secondLevelCacheStatistics =
        statistics.getSecondLevelCacheStatistics( "query.cache.person" );
long hitCount = secondLevelCacheStatistics.getHitCount();
long missCount = secondLevelCacheStatistics.getMissCount();
double hitRatio = (double) hitCount / ( hitCount + missCount );
```
JAVA

13.9. JCache

> Use of the build-in integration for JCache (https://jcp.org/en/jsr/detail?id=107) requires that the `hibernate-jcache` module jar (and all of its dependencies) are on the classpath. In addition a JCache implementation needs to be added as well. A list of compatible implementations can be found on the JCP website (https://jcp.org/aboutJava/communityprocess/implementations/jsr107/index.html). An alternative source of compatible implementations can be found through the JSR-107 test zoo (https://github.com/cruftex/jsr107-test-zoo).

13.9.1. RegionFactory

The `hibernate-jcache` module defines the following region factory: `JCacheRegionFactory`.

To use the `JCacheRegionFactory`, you need to specify the following configuration property:

Example 369. `JCacheRegionFactory` configuration

```xml
<property
    name="hibernate.cache.region.factory_class"
    value="org.hibernate.cache.jcache.JCacheRegionFactory"/>
```

The `JCacheRegionFactory` configures a `javax.cache.CacheManager`.

13.9.2. JCache `CacheManager`

JCache mandates that `CacheManager`s sharing the same URI and class loader be unique in JVM.

If you do not specify additional properties, the `JCacheRegionFactory` will load the default JCache provider and create the default `CacheManager`. Also, `Cache`s will be created using the default `javax.cache.configuration.MutableConfiguration`.

In order to control which provider to use and specify configuration for the `CacheManager` and `Cache`s you can use the following two properties:

Example 370. JCache configuration

```xml
<property
    name="hibernate.javax.cache.provider"
    value="org.ehcache.jsr107.EhcacheCachingProvider"/>
<property
    name="hibernate.javax.cache.uri"
    value="file:/path/to/ehcache.xml"/>
```

Only by specifying the second property `hibernate.javax.cache.uri` will you be able to have a `CacheManager` per `SessionFactory`.

13.10. Ehcache

This integration covers Ehcache 2.x, in order to use Ehcache 3.x as second level cache, refer to the JCache integration.

> Use of the build-in integration for Ehcache (http://www.ehcache.org/) requires that the `hibernate-ehcache` module jar (and all of its dependencies) are on the classpath.

13.10.1. RegionFactory

The hibernate-ehcache module defines two specific region factories: `EhCacheRegionFactory` and `SingletonEhCacheRegionFactory`.

`EhCacheRegionFactory`

To use the `EhCacheRegionFactory`, you need to specify the following configuration property:

Example 371. `EhCacheRegionFactory` *configuration*

```xml
<property
    name="hibernate.cache.region.factory_class"
    value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
```

The `EhCacheRegionFactory` configures a `net.sf.ehcache.CacheManager` for each `SessionFactory`, so the `CacheManager` is not shared among multiple `SessionFactory` instances in the same JVM.

`SingletonEhCacheRegionFactory`

To use the `SingletonEhCacheRegionFactory`, you need to specify the following configuration property:

Example 372. `SingletonEhCacheRegionFactory` *configuration*

```xml
<property
    name="hibernate.cache.region.factory_class"
    value="org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory"/>
```

The `SingletonEhCacheRegionFactory` configures a singleton `net.sf.ehcache.CacheManager` (see CacheManager#create() (http://www.ehcache.org/apidocs/2.8.4/net/sf/ehcache/CacheManager.html#create%28%29)), shared among multiple `SessionFactory` instances in the same JVM.

> Ehcache documentation (http://www.ehcache.org/documentation/2.8/integrations/hibernate#optional) recommends using multiple non-singleton `CacheManager(s)` when there are multiple Hibernate `SessionFactory` instances running in the same JVM.

13.11. Infinispan

> Use of the build-in integration for Infinispan (http://infinispan.org/) requires that the `hibernate-infinispan module` jar (and all of its dependencies) are on the classpath.

How to configure Infinispan to be the second level cache provider varies slightly depending on the deployment scenario:

13.11.1. Single Node Local

In standalone library mode, a JPA/Hibernate application runs inside a Java SE application or inside containers that don't offer Infinispan integration.

Enabling Infinispan second level cache provider inside a JPA/Hibernate application that runs in single node is very straightforward. First, make sure the Hibernate Infinispan cache provider (and its dependencies) are available in the classpath, then modify the persistence.xml to include these properties:

XML

```
<!-- Use Infinispan second level cache provider -->
<property name="hibernate.cache.region.factory_class"
          value="org.hibernate.cache.infinispan.InfinispanRegionFactory"/>

<!-- Optional: Force using local configuration when only using a single node.
     Otherwise a clustered configuration is loaded. -->
<property name="hibernate.cache.infinispan.cfg"
          value="org/hibernate/cache/infinispan/builder/infinispan-configs-local.xml"/>
```

Plugging in Infinispan as second-level cache provider requires at the bare minimum that `hibernate.cache.region.factory_class` is set to an Infinispan region factory implementation. Normally, this is `org.hibernate.cache.infinispan.InfinispanRegionFactory` but other region factories are possible in alternative scenarios (see Alternative Region Factory section for more info).

By default, the Infinispan second-level cache provider uses an Infinispan configuration that's designed for clustered environments. However, since this section is focused on running Infinispan second-level cache provider in a single node, an Infinispan configuration designed for local environments is recommended. To enable that configuration, set

`hibernate.cache.infinispan.cfg` to `org/hibernate/cache/infinispan/builder/infinispan-configs-local.xml` value.

The next section focuses on analysing how the default local configuration works. Changing Infinispan configuration options can be done following the instructions in Configuration Properties section.

Default Local Configuration

Infinispan second-level cache provider comes with a configuration designed for local, single node, environments. These are the characteristics of such configuration:

- Entities, collections, queries and timestamps are stored in non-transactional local caches.
- Entities and collections query caches are configured with the following eviction settings. You can change these settings on a per entity or collection basis or per individual entity or collection type. More information in the Advanced Configuration section below.
  - Eviction wake up interval is 5 seconds.
  - Max number of entries are 10,000
  - Max idle time before expiration is 100 seconds
  - Default eviction algorithm is LRU, least recently used.
- *No eviction/expiration is configured for timestamp caches*, nor it's allowed.

Local Cache Strategies

Before version 5.0, Infinispan only supported `transactional` and `read-only` strategies.

Starting with version 5.0, all cache strategies are supported: `transactional`, `read-write`, `nonstrict-read-write` and `read-only`.

13.11.2. Multi Node Cluster

When running a JPA/Hibernate in a multi-node environment and enabling Infinispan second-level cache, it is necessary to cluster the second-level cache so that cache consistency can be guaranteed. Clustering the Infinispan second-level cache provider is as simple as adding the following properties:

```
XML
<!-- Use Infinispan second level cache provider -->
<property name="hibernate.cache.region.factory_class"
          value="org.hibernate.cache.infinispan.InfinispanRegionFactory"/>
```

As with the standalone local mode, at the bare minimum the region factory has to be configured to point to an Infinispan region factory implementation.

However, the default Infinispan configuration used by the second-level cache provider is already configured to work in a cluster environment, so no need to add any extra properties.

The next section focuses on analysing how the default cluster configuration works. Changing Infinispan configuration options can be done following the instructions in Configuration Properties section.

Default Cluster Configuration

Infinispan second-level cache provider default configuration is designed for multi-node clustered environments. The aim of this section is to explain the default settings for each of the different global data type caches (entity, collection, query and timestamps), why these were chosen and what are the available alternatives. These are the characteristics of such configuration:

- For all entities and collections, whenever a new *entity or collection is read from database* and needs to be cached, *it's only cached locally* in order to reduce intra-cluster traffic. This option can be changed so that entities/collections are cached cluster wide, by switching the entity/collection cache to be replicated or distributed. How to change this option is explained in the Configuration Properties section.
- All *entities and collections are configured to use a synchronous invalidation* as clustering mode. This means that when an entity is updated, the updated cache will send a message to the other members of the cluster telling them that the entity has been modified. Upon receipt of this message, the other nodes will remove this data from their local cache, if it was stored there. This option can be changed so that both local and remote nodes contain the updates by configuring entities or collections to use a replicated or distributed cache. With replicated caches all nodes would contain the update, whereas with distributed caches only a subset of the nodes. How to change this option is explained in the Configuration Properties section.
- All *entities and collections have initial state transfer disabled* since there's no need for it.
- Entities and collections are configured with the following eviction settings. You can change these settings on a per entity or collection basis or per individual entity or collection type. More information in the Configuration Properties section below.
  - Eviction wake up interval is 5 seconds.
  - Max number of entries are 10,000
  - Max idle time before expiration is 100 seconds
  - Default eviction algorithm is LRU, least recently used.
- Assuming that query caching has been enabled for the persistence unit (see query cache section), the query cache is configured so that *queries are only cached locally*. Alternatively, you can configure query caching to use replication by selecting the `replicated-query` as query cache name. However, replication for query cache only makes sense if, and only if, all of this conditions are true:
  - Performing the query is quite expensive.
  - The same query is very likely to be repeatedly executed on different cluster nodes.
  - The query is unlikely to be invalidated out of the cache (Note: Hibernate must aggressively invalidate query results from the cache any time any instance of one of the entity types targeted by the query. All such query results are invalidated, even if the change made to the specific entity instance would not have affected the query result)

- *query cache* uses the *same eviction/expiration settings as for entities/collections*.
- *query cache has initial state transfer disabled* . It is not recommended that this is enabled.
- The *timestamps cache is configured with asynchronous replication* as clustering mode. Local or invalidated cluster modes are not allowed, since all cluster nodes must store all timestamps. As a result, *no eviction/expiration is allowed for timestamp caches either*.

> Asynchronous replication was selected as default for timestamps cache for performance reasons. A side effect of this choice is that when an entity/collection is updated, for a very brief period of time stale queries might be returned. It's important to note that due to how Infinispan deals with asynchronous replication, stale queries might be found even query is done right after an entity/collection update on same node. The reason why asynchronous replication works this way is because there's a single node that's owner for a given key, and that enables changes to be applied in the same order in all nodes. Without it, it could happen that an older value could replace a newer value in certain nodes.

> Hibernate must aggressively invalidate query results from the cache any time any instance of one of the entity types is modified. All cached query results referencing given entity type are invalidated, even if the change made to the specific entity instance would not have affected the query result. The timestamps cache plays here an important role - it contains last modification timestamp for each entity type. After a cached query results is loaded, its timestamp is compared to all timestamps of the entity types that are referenced in the query and if any of these is higher, the cached query result is discarded and the query is executed against DB.

Cluster Cache Strategies

Before version 5.0, Infinispan only supported `transactional` and `read-only` strategies on top of *transactional invalidation* caches.

Since version 5.0, Infinispan currently supports all cache concurrency modes in cluster mode, although not all combinations of configurations are compatible:

- *non-transactional invalidation* caches are supported as well with `read-write` strategy. The actual setting of cache concurrency mode (`read-write` vs. `transactional`) is not honored, the appropriate strategy is selected based on the cache configuration (*non-transactional* vs. *transactional*).
- `read-write` mode is supported on *non-transactional distributed/replicated* caches, however, eviction should not be used in this configuration. Use of eviction can lead to consistency issues. Expiration (with reasonably long max-idle times) can be used.
- `nonstrict-read-write` mode is supported on *non-transactional distributed/replicated* caches, but the eviction should be turned off as well. In addition to that, the entities must use versioning. This mode mildly relaxes the consistency - between DB commit and end of transaction commit a stale read (see example) may occur in another transaction. However this strategy uses less RPCs and can be more performant than the other ones.
- `read-only` mode is supported on both *transactional* and *non-transactional invalidation* caches and *non-transactional distributed/replicated* caches, but use of this mode currently does not bring any performance gains.

The available combinations are summarized in table below:

Table 6. Cache concurrency strategy/cache mode compatibility table

Concurrency strategy	Cache transactions	Cache mode	Eviction
transactional	transactional	invalidation	yes
read-write	non-transactional	invalidation	yes
read-write	non-transactional	distributed/replicated	no
nonstrict-read-write	non-transactional	distributed/replicated	no

Changing caches to behave different to the default behaviour explained in previous section is explained in Configuration Properties section.

Example 373. Stale read with `nonstrict-read-write` strategy

```
A=0 (non-cached), B=0 (cached in 2LC)
TX1: write A = 1, write B = 1
TX1: start commit
TX1: commit A, B in DB
TX2: read A = 1 (from DB), read B = 0 (from 2LC) // breaks transactional atomicity
TX1: update A, B in 2LC
TX1: end commit
Tx3: read A = 1, B = 1 // reads after TX1 commit completes are consistent again
```

13.11.3. Alternative RegionFactory

In standalone environments or managed environments with no Infinispan integration, `org.hibernate.cache.infinispan.InfinispanRegionFactory` should be the choice for region factory implementation. However, it might be sometimes desirable for the Infinispan cache manager to be shared between different JPA/Hibernate applications, for example to share intra-cluster communications channels. In this case, the Infinispan cache manager could be bound into JNDI and the JPA/Hibernate applications could use an alternative region factory implementation:

Example 374. `JndiInfinispanRegionFactory` *configuration*

```xml
<property
    name="hibernate.cache.region.factory_class"
    value="org.hibernate.cache.infinispan.JndiInfinispanRegionFactory" />

<property
    name="hibernate.cache.infinispan.cachemanager"
    value="java:CacheManager" />
```

13.11.4. Inside Wildfly

In WildFly, Infinispan is the default second level cache provider for JPA/Hibernate. When using JPA in WildFly, region factory is automatically set upon configuring `hibernate.cache.use_second_level_cache=true` (by default second-level cache is not used).

You can find details about its configuration in the JPA reference guide, in particular, in the second level cache section.

The default second-level cache configurations used by Wildfly match the configurations explained above both for local and clustered environments. So, an Infinispan based second-level cache should behave exactly the same standalone and within containers that provide Infinispan second-level cache as default for JPA/Hibernate.

> Remember that if deploying to Wildfly or Application Server, the way some Infinispan second level cache provider configuration is defined changes slightly because the properties must include deployment and persistence information. Check the Configuration section for more details.

13.11.5. Configuration properties

As explained above, Infinispan second-level cache provider comes with default configuration in `infinispan-config.xml` that is suited for clustered use. If there's only single JVM accessing the DB, you can use more performant `infinispan-config-local.xml` by setting the `hibernate.cache.infinispan.cfg` property. If you require further tuning of the cache, you can

provide your own configuration. Caches that are not specified in the provided configuration will default to `infinispan-config.xml` (if the provided configuration uses clustering) or `infinispan-config-local.xml`.

> It is not possible to specify the configuration this way in WildFly. Cache configuration changes in Wildfly should be done either modifying the cache configurations inside the application server configuration, or creating new caches with the desired tweaks and plugging them accordingly. See examples below on how entity/collection specific configurations can be applied.

Example 375. Use custom Infinispan configuration

```xml
<property
    name="hibernate.cache.infinispan.cfg"
    value="my-infinispan-configuration.xml" />
```

> If the cache is configured as transactional, InfinispanRegionFactory automatically sets transaction manager so that the TM used by Infinispan is the same as TM used by Hibernate.

Cache configuration can differ for each type of data stored in the cache. In order to override the cache configuration template, use property `hibernate.cache.infinispan.`*`data-type`*`.cfg` where *`data-type`* can be one of:

`entity`
Entities indexed by `@Id` or `@EmbeddedId` attribute.

`immutable-entity`
Entities tagged with `@Immutable` annotation or set as `mutable=false` in mapping file.

`naturalid`

Entities indexed by their `@NaturalId` attribute.

`collection`

All collections.

`timestamps`

Mapping *entity type → last modification timestamp*. Used for query caching.

`query`

Mapping *query → query result*.

`pending-puts`

Auxiliary caches for regions using invalidation mode caches.

For specifying cache template for specific region, use region name instead of the *`data-type`*:

Example 376. Use custom cache template

```xml
<property
    name="hibernate.cache.infinispan.entities.cfg"
    value="custom-entities" />
<property
    name="hibernate.cache.infinispan.query.cfg"
    value="custom-query-cache" />
<property
    name="hibernate.cache.infinispan.com.example.MyEntity.cfg"
    value="my-entities" />
<property
    name="hibernate.cache.infinispan.com.example.MyEntity.someCollection.cfg"
    value="my-entities-some-collection" />
```

Use custom cache template in Wildfly

When applying entity/collection level changes inside JPA applications deployed in Wildfly, it is necessary to specify deployment name and persistence unit name:

```xml
<property
    name="hibernate.cache.infinispan._war_or_ear_name_._unit_name_.com.example.MyEntity.cfg"
    value="my-entities" />
<property
    name="hibernate.cache.infinispan._war_or_ear_name_._unit_name_.com.example.MyEntity.someCollection.cfg"
    value="my-entities-some-collection" />
```

> Cache configurations are used only as a template for the cache created for given region (usually each entity hierarchy or collection has its own region). It is not possible to use the same cache for different regions.

Some options in the cache configuration can also be overridden directly through properties. These are:

`hibernate.cache.infinispan.`*`something`*`.eviction.strategy`
Available options are `NONE`, `LRU` and `LIRS`.

`hibernate.cache.infinispan.`*`something`*`.eviction.max_entries`
Maximum number of entries in the cache.

`hibernate.cache.infinispan.`*`something`*`.expiration.lifespan`
Lifespan of entry from insert into cache (in milliseconds)

`hibernate.cache.infinispan.`*`something`*`.expiration.max_idle`
Lifespan of entry from last read/modification (in milliseconds)

`hibernate.cache.infinispan.`*`something`*`.expiration.wake_up_interval`
Period of thread checking expired entries.

`hibernate.cache.infinispan.statistics`
Globally enables/disable Infinispan statistics collection, and their exposure via JMX.

Example:

```xml
<property name="hibernate.cache.infinispan.entity.eviction.strategy"
   value= "LRU"/>
<property name="hibernate.cache.infinispan.entity.eviction.wake_up_interval"
   value= "2000"/>
<property name="hibernate.cache.infinispan.entity.eviction.max_entries"
   value= "5000"/>
<property name="hibernate.cache.infinispan.entity.expiration.lifespan"
   value= "60000"/>
<property name="hibernate.cache.infinispan.entity.expiration.max_idle"
   value= "30000"/>
```

With the above configuration, you're overriding whatever eviction/expiration settings were defined for the default entity cache name in the Infinispan cache configuration used, regardless of whether it's the default one or user defined. More specifically, we're defining the following:

- All entities to use LRU eviction strategy
- The eviction thread to wake up every 2 seconds (2000 milliseconds)
- The maximum number of entities for each entity type to be 5000 entries
- The lifespan of each entity instance to be 1 minute (600000 milliseconds).
- The maximum idle time for each entity instance to be 30 seconds (30000 milliseconds).

You can also override eviction/expiration settings on a per entity/collection type basis in such way that the overriden settings only afftect that particular entity (i.e. `com.acme.Person`) or collection type (i.e. `com.acme.Person.addresses`). Example:

```xml
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.strategy"
   value= "LIRS"/>
```

Inside of Wildfly, same as with the entity/collection configuration override, eviction/expiration settings would also require deployment name and persistence unit information:

```xml
<property name="hibernate.cache.infinispan._war_or_ear_name_._unit_name_.com.acme.Person.eviction.strategy"
   value= "LIRS"/>
<property name="hibernate.cache.infinispan._war_or_ear_name_._unit_name_.com.acme.Person.expiration.lifespan"
   value= "65000"/>
```

In versions prior to 5.1, `hibernate.cache.infinispan.*something*.expiration.wake_up_interval` was called `hibernate.cache.infinispan.*something*.eviction.wake_up_interval`. Eviction settings are checked upon each cache insert, it is expiration that needs to be triggered periodically. The old property still works, but its use is deprecated.

Property `hibernate.cache.infinispan.use_synchronization` that allowed to register Infinispan as XA resource in the transaction has been deprecated in 5.0 and is not honored anymore. Infinispan 2LC must register as synchronizations on transactional caches. Also, non-transactional cache modes hook into the current JTA/JDBC transaction as synchronizations.

13.11.6. Remote Infinispan Caching

Lately, several questions (here (http://community.jboss.org/message/575814#575814) and here (http://community.jboss.org/message/585841#585841)) have appeared in the Infinispan user forums asking whether it'd be possible to have an Infinispan second level cache that instead of living in the same JVM as the Hibernate code, it resides in a remote server, i.e. an Infinispan Hot Rod server. It's important to understand that trying to set up second level cache in this way is generally not a good idea for the following reasons:

- The purpose of a JPA/Hibernate second level cache is to store entities/collections recently retrieved from database or to maintain results of recent queries. So, part of the aim of the second level cache is to have data accessible locally rather than having to go to the database to retrieve it everytime this is needed. Hence, if you decide to set the second level cache to be remote as well, you're losing one of the key advantages of the second level cache: the fact that the cache is local to the code that requires it.
- Setting a remote second level cache can have a negative impact in the overall performance of your application because it means that cache misses require accessing a remote location to verify whether a particular entity/collection/query is cached. With a local second level cache however, these misses are resolved locally and so they are much faster to execute than with a remote second level cache.

There are however some edge cases where it might make sense to have a remote second level cache, for example:

- You are having memory issues in the JVM where JPA/Hibernate code and the second level cache is running. Off loading the second level cache to remote Hot Rod servers could be an interesting way to separate systems and allow you find the culprit of the memory issues more easily.
- Your application layer cannot be clustered but you still want to run multiple application layer nodes. In this case, you can't have multiple local second level cache instances running because they won't be able to invalidate each other for example when data in the second level cache is updated. In this case, having a remote second level cache could be a way out to make sure your second level cache is always in a consistent state, will all nodes in the application layer pointing to it.
- Rather than having the second level cache in a remote server, you want to simply keep the cache in a separate VM still within the same machine. In this case you would still have the additional overhead of talking across to another JVM, but it wouldn't have the latency of across a network.

  The benefit of doing this is that:

  - Size the cache separate from the application, since the cache and the application server have very different memory profiles. One has lots of short lived objects, and the other could have lots of long lived objects.
  - To pin the cache and the application server onto different CPU cores (using *numactl*), and even pin them to different physically memory based on the NUMA nodes.

14. Interceptors and events

It is useful for the application to react to certain events that occur inside Hibernate. This allows for the implementation of generic functionality and the extension of Hibernate functionality.

14.1. Interceptors

The `org.hibernate.Interceptor` interface provides callbacks from the session to the application, allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded.

One possible use for this is to track auditing information. The following example shows an `Interceptor` implementation that automatically logs when an entity is updated.

JAVA

```
public static class LoggingInterceptor extends EmptyInterceptor {
    @Override
    public boolean onFlushDirty(
        Object entity,
        Serializable id,
        Object[] currentState,
        Object[] previousState,
        String[] propertyNames,
        Type[] types) {
            LOGGER.debugv( "Entity {0}#{1} changed from {2} to {3}",
                entity.getClass().getSimpleName(),
                id,
                Arrays.toString( previousState ),
                Arrays.toString( currentState )
            );
            return super.onFlushDirty( entity, id, currentState,
                previousState, propertyNames, types
        );
    }
}
```

> You can either implement `Interceptor` directly or extend `org.hibernate.EmptyInterceptor`.

An Interceptor can be either `Session`-scoped or `SessionFactory`-scoped.

A Session-scoped interceptor is specified when a session is opened.

```java
SessionFactory sessionFactory = entityManagerFactory.unwrap( SessionFactory.class );
Session session = sessionFactory
    .withOptions()
    .interceptor(new LoggingInterceptor() )
    .openSession();
session.getTransaction().begin();

Customer customer = session.get( Customer.class, customerId );
customer.setName( "Mr. John Doe" );
//Entity Customer#1 changed from [John Doe, 0] to [Mr. John Doe, 0]

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

A `SessionFactory`-scoped interceptor is registered with the `Configuration` object prior to building the `SessionFactory`. Unless a session is opened explicitly specifying the interceptor to use, the `SessionFactory`-scoped interceptor will be applied to all sessions opened from that `SessionFactory`. `SessionFactory`-scoped interceptors must be thread safe. Ensure that you do not store session-specific states since multiple sessions will use this interceptor potentially concurrently.

```java
SessionFactory sessionFactory = new MetadataSources( new StandardServiceRegistryBuilder().build() )
    .addAnnotatedClass( Customer.class )
    .getMetadataBuilder()
    .build()
    .getSessionFactoryBuilder()
    .applyInterceptor( new LoggingInterceptor() )
    .build();
```

14.2. Native Event system

If you have to react to particular events in the persistence layer, you can also use the Hibernate *event* architecture. The event system can be used in place of or in addition to interceptors.

Many methods of the `Session` interface correlate to an event type. The full range of defined event types is declared as enum values on `org.hibernate.event.spi.EventType`. When a request is made of one of these methods, the Session generates an appropriate event and passes it to the configured event listener(s) for that type.

Applications are free to implement a customization of one of the listener interfaces (i.e., the `LoadEvent` is processed by the registered implementation of the `LoadEventListener` interface), in which case their implementation would be responsible for processing any `load()` requests made of the `Session`.

> The listeners should be considered stateless; they are shared between requests, and should not save any state as instance variables.

A custom listener implements the appropriate interface for the event it wants to process and/or extend one of the convenience base classes (or even the default event listeners used by Hibernate out-of-the-box as these are declared non-final for this purpose).

Here is an example of a custom load event listener:

Example 377. Custom `LoadListener` example

```
JAVA
EntityManagerFactory entityManagerFactory = entityManagerFactory();
SessionFactoryImplementor sessionFactory = entityManagerFactory.unwrap( SessionFactoryImplementor.class );
sessionFactory
    .getServiceRegistry()
    .getService( EventListenerRegistry.class )
    .prependListeners( EventType.LOAD, new SecuredLoadEntityListener() );

Customer customer = entityManager.find( Customer.class, customerId );
```

14.3. Mixing Events and Interceptors

When you want to customize the entity state transition behavior, you have to options:

1. you provide a custom `Interceptor`, which is taken into consideration by the default Hibernate event listeners. For example, the `Interceptor#onSave()` method is invoked by Hibernate `AbstractSaveEventListener`. Or, the `Interceptor#onLoad()` is called by the `DefaultPreLoadEventListener`.
2. you can replace any given default event listener with your own implementation. When doing this, you should probably extend the default listeners because otherwise you'd have to take care of all the low-level entity state transition logic. For example, if you replace the `DefaultPreLoadEventListener` with your own implementation, then, only if you call the `Interceptor#onLoad()` method explicitly, you can mix the custom load event listener with a custom Hibernate interceptor.

14.4. Hibernate declarative security

Usually, declarative security in Hibernate applications is managed in a session facade layer. Hibernate allows certain actions to be authorized via JACC and JAAS. This is an optional functionality that is built on top of the event architecture.

First, you must configure the appropriate event listeners, to enable the use of JACC authorization. Again, see Event Listener Registration for the details.

Below is an example of an appropriate `org.hibernate.integrator.spi.Integrator` implementation for this purpose.

Example 378. JACC listener registration example

JAVA

```
public static class JaccIntegrator implements ServiceContributingIntegrator {

    private static final Logger log = Logger.getLogger( JaccIntegrator.class );

    private static final DuplicationStrategy DUPLICATION_STRATEGY =
            new DuplicationStrategy() {
        @Override
        public boolean areMatch(Object listener, Object original) {
            return listener.getClass().equals( original.getClass() ) &&
                    JaccSecurityListener.class.isInstance( original );
        }

        @Override
        public Action getAction() {
            return Action.KEEP_ORIGINAL;
        }
    };

    @Override
    public void prepareServices(
            StandardServiceRegistryBuilder serviceRegistryBuilder) {
        boolean isSecurityEnabled = serviceRegistryBuilder
                .getSettings().containsKey( AvailableSettings.JACC_ENABLED );
        final JaccService jaccService = isSecurityEnabled ?
                new StandardJaccServiceImpl() : new DisabledJaccServiceImpl();
        serviceRegistryBuilder.addService( JaccService.class, jaccService );
    }

    @Override
    public void integrate(
            Metadata metadata,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        doIntegration(
                serviceRegistry
                        .getService( ConfigurationService.class ).getSettings(),
                // pass no permissions here, because atm actually injecting the
                // permissions into the JaccService is handled on SessionFactoryImpl via
                // the org.hibernate.boot.cfgxml.spi.CfgXmlAccessService
                null,
                serviceRegistry
        );
    }

    private void doIntegration(
            Map properties,
            JaccPermissionDeclarations permissionDeclarations,
            SessionFactoryServiceRegistry serviceRegistry) {
        boolean isSecurityEnabled = properties
                .containsKey( AvailableSettings.JACC_ENABLED );
        if ( ! isSecurityEnabled ) {
            log.debug( "Skipping JACC integration as it was not enabled" );
            return;
        }

        final String contextId = (String) properties
```

```
                .get( AvailableSettings.JACC_CONTEXT_ID );
        if ( contextId == null ) {
            throw new IntegrationException( "JACC context id must be specified" );
        }

        final JaccService jaccService = serviceRegistry
                .getService( JaccService.class );
        if ( jaccService == null ) {
            throw new IntegrationException( "JaccService was not set up" );
        }

        if ( permissionDeclarations != null ) {
            for ( GrantedPermission declaration : permissionDeclarations
                    .getPermissionDeclarations() ) {
                jaccService.addPermission( declaration );
            }
        }

        final EventListenerRegistry eventListenerRegistry =
                serviceRegistry.getService( EventListenerRegistry.class );
        eventListenerRegistry.addDuplicationStrategy( DUPLICATION_STRATEGY );

        eventListenerRegistry.prependListeners(
                EventType.PRE_DELETE, new JaccPreDeleteEventListener() );
        eventListenerRegistry.prependListeners(
                EventType.PRE_INSERT, new JaccPreInsertEventListener() );
        eventListenerRegistry.prependListeners(
                EventType.PRE_UPDATE, new JaccPreUpdateEventListener() );
        eventListenerRegistry.prependListeners(
                EventType.PRE_LOAD, new JaccPreLoadEventListener() );
    }

    @Override
    public void disintegrate(SessionFactoryImplementor sessionFactory,
                             SessionFactoryServiceRegistry serviceRegistry) {
        // nothing to do
    }
}
```

You must also decide how to configure your JACC provider. Consult your JACC provider documentation.

14.5. JPA Callbacks

JPA also defines a more limited set of callbacks through annotations.

Table 7. Callback annotations

Type	Description
@PrePersist	Executed before the entity manager persist operation is actually executed or cascaded. This call is synchronous with the persist operation.

Type	Description
@PreRemove	Executed before the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
@PostPersist	Executed after the entity manager persist operation is actually executed or cascaded. This call is invoked after the database INSERT is executed.
@PostRemove	Executed after the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
@PreUpdate	Executed before the database UPDATE operation.
@PostUpdate	Executed after the database UPDATE operation.
@PostLoad	Executed after an entity has been loaded into the current persistence context or an entity has been refreshed.

There are two available approaches defined for specifying callback handling:

- The first approach is to annotate methods on the entity itself to receive notification of particular entity life cycle event(s).
- The second is to use a separate entity listener class. An entity listener is a stateless class with a no-arg constructor. The callback annotations are placed on a method of this class instead of the entity class. The entity listener class is then associated with the entity using the `javax.persistence.EntityListeners` annotation

Example 379. Example of specifying JPA callbacks

JAVA

```
@Entity
@EntityListeners( LastUpdateListener.class )
public static class Person {

    @Id
    private Long id;

    private String name;

    private Date dateOfBirth;

    @Transient
    private long age;

    private Date lastUpdate;

    public void setLastUpdate(Date lastUpdate) {
        this.lastUpdate = lastUpdate;
    }

    /**
     * Set the transient property at load time based on a calculation.
     * Note that a native Hibernate formula mapping is better for this purpose.
     */
    @PostLoad
    public void calculateAge() {
        age = ChronoUnit.YEARS.between( LocalDateTime.ofInstant(
                Instant.ofEpochMilli( dateOfBirth.getTime()), ZoneOffset.UTC),
            LocalDateTime.now()
        );
    }
}

public static class LastUpdateListener {

    @PreUpdate
    @PrePersist
    public void setLastUpdate( Person p ) {
        p.setLastUpdate( new Date() );
    }
}
```

These approaches can be mixed, meaning you can use both together.

Regardless of whether the callback method is defined on the entity or on an entity listener, it must have a void-return signature. The name of the method is irrelevant as it is the placement of the callback annotations that makes the method a callback. In the case of callback methods defined on the entity class, the method must additionally have a no-argument signature. For callback methods defined on an entity listener class, the method must have a single argument signature; the type of that argument can be either `java.lang.Object` (to facilitate attachment to multiple entities) or the specific entity type.

A callback method can throw a `RuntimeException`. If the callback method does throw a `RuntimeException`, then the current transaction, if any, must be rolled back.

A callback method must not invoke `EntityManager` or `Query` methods!

It is possible that multiple callback methods are defined for a particular lifecycle event. When that is the case, the defined order of execution is well defined by the JPA spec (specifically section 3.5.4):

- Any default listeners associated with the entity are invoked first, in the order they were specified in the XML. See the `javax.persistence.ExcludeDefaultListeners` annotation.
- Next, entity listener class callbacks associated with the entity hierarchy are invoked, in the order they are defined in the `EntityListeners`. If multiple classes in the entity hierarchy define entity listeners, the listeners defined for a superclass are invoked before the listeners defined for its subclasses. See the `javax.persistence.ExcludeSuperclassListener`s annotation.
- Lastly, callback methods defined on the entity hierarchy are invoked. If a callback type is annotated on both an entity and one or more of its superclasses without method overriding, both would be called, the most general superclass first. An entity class is also allowed to override a callback method defined in a superclass in which case the super callback would not get invoked; the overriding method would get invoked provided it is annotated.

15. HQL and JPQL

The Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object model focused query languages similar in nature to SQL. JPQL is a heavily-inspired-by subset of HQL. A JPQL query is always a valid HQL query, the reverse is not true however.

Both HQL and JPQL are non-type-safe ways to perform query operations. Criteria queries offer a type-safe approach to querying. See Criteria for more information.

15.1. Query API

15.2. Examples domain model

To better understand the further HQL and JPQL examples, it's time to familiarize the domain model entities that are used in all the examples features in this chapter.

Example 380. Examples domain model

JAVA

```
@NamedQueries(
    @NamedQuery(
        name = "get_person_by_name",
        query = "select p from Person p where name = :name"
    )
)
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String nickName;

    private String address;

    @Temporal(TemporalType.TIMESTAMP )
    private Date createdOn;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    @OrderColumn(name = "order_id")
    private List<Phone> phones = new ArrayList<>();

    @ElementCollection
    @MapKeyEnumerated(EnumType.STRING)
    private Map<AddressType, String> addresses = new HashMap<>();

    @Version
    private int version;

    //Getters and setters are omitted for brevity

}

public enum AddressType {
    HOME,
    OFFICE
}

@Entity
public class Partner {

        @Id
        @GeneratedValue
        private Long id;

        private String name;

        @Version
        private int version;

        //Getters and setters are omitted for brevity
```

```
}

@Entity
public class Phone {

    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;

    @Column(name = "phone_number")
    private String number;

    @Enumerated(EnumType.STRING)
    @Column(name = "phone_type")
    private PhoneType type;

    @OneToMany(mappedBy = "phone", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Call> calls = new ArrayList<>(  );

    @OneToMany(mappedBy = "phone")
    @MapKey(name = "timestamp")
    @MapKeyTemporal(TemporalType.TIMESTAMP )
    private Map<Date, Call> callHistory = new HashMap<>();

    @ElementCollection
    private List<Date> repairTimestamps = new ArrayList<>(  );

    //Getters and setters are omitted for brevity

}

public enum PhoneType {
    LAND_LINE,
    MOBILE;
}

@Entity
@Table(name = "phone_call")
public class Call {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private Phone phone;

    @Column(name = "call_timestamp")
    private Date timestamp;

    private int duration;

    //Getters and setters are omitted for brevity

}
```

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

    @Id
    @GeneratedValue
    private Long id;

    private BigDecimal amount;

    private boolean completed;

    @ManyToOne
    private Person person;

    //Getters and setters are omitted for brevity

}

@Entity
public class CreditCardPayment extends Payment {
}

@Entity
public class WireTransferPayment extends Payment {
}
```

15.3. JPA Query API

In JPA the query is represented by `javax.persistence.Query` or `javax.persistence.TypedQuery` as obtained from the `EntityManager`. The create an inline `Query` or `TypedQuery`, you need to use the `EntityManager#createQuery` method. For named queries, the `EntityManager#createNamedQuery` method is needed.

Example 381. Obtaining a JPA `Query` or a `TypedQuery` reference

JAVA

```
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name"
);

TypedQuery<Person> typedQuery = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name", Person.class
);
```

Example 382. Obtaining a JPA `Query` or a `TypedQuery` reference for a named query

```java
@NamedQueries(
    @NamedQuery(
        name = "get_person_by_name",
        query = "select p from Person p where name = :name"
    )
)

Query query = entityManager.createNamedQuery( "get_person_by_name" );

TypedQuery<Person> typedQuery = entityManager.createNamedQuery(
    "get_person_by_name", Person.class
);
```

The `Query` interface can then be used to control the execution of the query. For example, we may want to specify an execution timeout or control caching.

Example 383. Basic JPA `Query` usage

```java
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
// timeout - in milliseconds
.setHint( "javax.persistence.query.timeout", 2000 )
// flush only at commit time
.setFlushMode( FlushModeType.COMMIT );
```

For complete details, see the `Query` Javadocs (http://docs.oracle.com/javaee/7/api/javax/persistence/Query.html). Many of the settings controlling the execution of the query are defined as hints. JPA defines some standard hints (like timeout in the example), but most are provider specific. Relying on provider specific hints limits your applications portability to some degree.

`javax.persistence.query.timeout`

Defines the query timeout, in milliseconds.

`javax.persistence.fetchgraph`

Defines a *fetchgraph* EntityGraph. Attributes explicitly specified as `AttributeNodes` are treated as `FetchType.EAGER` (via join fetch or subsequent select). For details, see the EntityGraph discussions in Fetching.

`javax.persistence.loadgraph`

Defines a *loadgraph* EntityGraph. Attributes explicitly specified as AttributeNodes are treated as `FetchType.EAGER` (via join fetch or subsequent select). Attributes that are not specified are treated as `FetchType.LAZY` or `FetchType.EAGER` depending on the attribute's definition in metadata. For details, see the EntityGraph discussions in Fetching.

`org.hibernate.cacheMode`

Defines the `CacheMode` to use. See `org.hibernate.query.Query#setCacheMode`.

`org.hibernate.cacheable`

Defines whether the query is cacheable. true/false. See `org.hibernate.query.Query#setCacheable`.

`org.hibernate.cacheRegion`

For queries that are cacheable, defines a specific cache region to use. See `org.hibernate.query.Query#setCacheRegion`.

`org.hibernate.comment`

Defines the comment to apply to the generated SQL. See `org.hibernate.query.Query#setComment`.

`org.hibernate.fetchSize`

Defines the JDBC fetch-size to use. See `org.hibernate.query.Query#setFetchSize`

`org.hibernate.flushMode`

Defines the Hibernate-specific `FlushMode` to use. See `org.hibernate.query.Query#setFlushMode`. If possible, prefer using `javax.persistence.Query#setFlushMode` instead.

`org.hibernate.readOnly`

Defines that entities and collections loaded by this query should be marked as read-only. See `org.hibernate.query.Query#setReadOnly`

The final thing that needs to happen before the query can be executed is to bind the values for any defined parameters. JPA defines a simplified set of parameter binding methods. Essentially it supports setting the parameter value (by name/position) and a specialized form for `Calendar` / `Date` types additionally accepting a `TemporalType`.

Example 384. JPA name parameter binding

```java
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" );

// For generic temporal field types (e.g. `java.util.Date`, `java.util.Calendar`)
// we also need to provide the associated `TemporalType`
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.createdOn > :timestamp" )
.setParameter( "timestamp", timestamp, TemporalType.DATE );
```

JPQL-style positional parameters are declared using a question mark followed by an ordinal - `?1`, `?2`. The ordinals start with 1. Just like with named parameters, positional parameters can also appear multiple times in a query.

Example 385. JPA positional parameter binding

```java
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like ?1" )
.setParameter( 1, "J%" );
```

> It's good practice not to mix forms in a given query.

In terms of execution, JPA `Query` offers 2 different methods for retrieving a result set.

- `Query#getResultList()` - executes the select query and returns back the list of results.
- `Query#getSingleResult()` - executes the select query and returns a single result. If there were more than one result an exception is thrown.

Example 386. JPA `getResultList()` result

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.getResultList();
```

JAVA

Example 387. JPA `getSingleResult()`

```
Person person = (Person) entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.getSingleResult();
```

JAVA

15.4. Hibernate Query API

In Hibernate, the HQL query is represented as `org.hibernate.query.Query` which is obtained from a `Session`. If the HQL is a named query, `Session#getNamedQuery` would be used; otherwise `Session#createQuery` is needed.

Example 388. Obtaining a Hibernate `Query`

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name"
);
```

JAVA

Example 389. Obtaining a Hibernate `Query` reference for a named query

```
org.hibernate.query.Query query = session.getNamedQuery( "get_person_by_name" );
```

JAVA

Not only was the JPQL syntax heavily inspired by HQL, but many of the JPA APIs were heavily inspired by Hibernate too. The two `Query` contracts are very similar.

The Query interface can then be used to control the execution of the query. For example, we may want to specify an execution timeout or control caching.

Example 390. Basic Query usage - Hibernate

```java
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
// timeout - in seconds
.setTimeout( 2 )
// write to L2 caches, but do not read from them
.setCacheMode( CacheMode.REFRESH )
// assuming query cache was enabled for the SessionFactory
.setCacheable( true )
// add a comment to the generated SQL if enabled via the hibernate.use_sql_comments configuration property
.setComment( "+ INDEX(p idx_person_name)" );
```

For complete details, see the Query (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Query.html) Javadocs.

Query hints here are database query hints. They are added directly to the generated SQL according to `Dialect#getQueryHintString`. The JPA notion of query hints, on the other hand, refer to hints that target the provider (Hibernate). So even though they are called the same, be aware they have a very different purpose. Also, be aware that Hibernate query hints generally make the application non-portable across databases unless the code adding them first checks the Dialect.

Flushing is covered in detail in Flushing. Locking is covered in detail in Locking. The concept of read-only state is covered in Persistence Contexts.

Hibernate also allows an application to hook into the process of building the query results via the `org.hibernate.transform.ResultTransformer` contract. See its Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/transform/ResultTransformer.html) as well as the Hibernate-provided implementations for additional details.

The last thing that needs to happen before we can execute the query is to bind the values for any parameters defined in the query. Query defines many overloaded methods for this purpose. The most generic form takes the value as well as the Hibernate Type.

Example 391. Hibernate name parameter binding

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%", StringType.INSTANCE );
```

JAVA

Hibernate generally understands the expected type of the parameter given its context in the query. In the previous example since we are using the parameter in a `LIKE` comparison against a String-typed attribute Hibernate would automatically infer the type; so the above could be simplified.

Example 392. Hibernate name parameter binding (inferred type)

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" );
```

JAVA

There are also short hand forms for binding common types such as strings, booleans, integers, etc.

Example 393. Hibernate name parameter binding (short forms)

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name " +
    "  and p.createdOn > :timestamp" )
.setParameter( "name", "J%" )
.setParameter( "timestamp", timestamp, TemporalType.TIMESTAMP);
```
JAVA

HQL-style positional parameters follow JDBC positional parameter syntax. They are declared using `?` without a following ordinal. There is no way to relate two such positional parameters as being "the same" aside from binding the same value to each.

Example 394. Hibernate positional parameter binding

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like ? " )
.setParameter( 0, "J%" );
```
JAVA

> This form should be considered deprecated and may be removed in the near future.

In terms of execution, Hibernate offers 4 different methods. The 2 most commonly used are

- `Query#list` - executes the select query and returns back the list of results.
- `Query#uniqueResult` - executes the select query and returns the single result. If there were more than one result an exception is thrown.

Example 395. Hibernate `list()` result

```
List<Person> persons = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.list();
```

JAVA

It is also possible to extract a single result from a `Query`.

Example 396. Hibernate `uniqueResult()`

```
Person person = (Person) session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.uniqueResult();
```

JAVA

> If the unique result is used often and the attributes upon which it is based are unique, you may want to consider mapping a natural-id and using the natural-id loading API. See the Natural Ids for more information on this topic.

15.4.1. Query scrolling

Hibernate offers additional, specialized methods for scrolling the query and handling results using a server-side cursor.

`Query#scroll` works in tandem with the JDBC notion of a scrollable `ResultSet`.

The `Query#scroll` method is overloaded:

- Then main form accepts a single argument of type `org.hibernate.ScrollMode` which indicates the type of scrolling to be used. See the Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/ScrollMode.html) for the details on each.
- The second form takes no argument and will use the `ScrollMode` indicated by `Dialect#defaultScrollMode`.

`Query#scroll` returns a `org.hibernate.ScrollableResults` which wraps the underlying JDBC (scrollable) `ResultSet` and provides access to the results. Unlike a typical forward-only `ResultSet`, the `ScrollableResults` allows you to navigate the `ResultSet` in any direction.

Example 397. Scrolling through a `ResultSet` containing entities

JAVA

```
try ( ScrollableResults scrollableResults = session.createQuery(
        "select p " +
        "from Person p " +
        "where p.name like :name" )
        .setParameter( "name", "J%" )
        .scroll()
) {
    while(scrollableResults.next()) {
        Person person = (Person) scrollableResults.get()[0];
        process(person);
    }
}
```

> Since this form holds the JDBC `ResultSet` open, the application should indicate when it is done with the `ScrollableResults` by calling its `close()` method (as inherited from `java.io.Closeable` so that `ScrollableResults` will work with try-with-resources (https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) blocks).

If left unclosed by the application, Hibernate will automatically close the underlying resources (e.g. `ResultSet` and `PreparedStatement`) used internally by the `ScrollableResults` when the current transaction is ended (either commit or rollback).

However, it is good practice to close the `ScrollableResults` explicitly.

> If you plan to use `Query#scroll` with collection fetches it is important that your query explicitly order the results so that the JDBC results contain the related rows sequentially.

Hibernate also supports `Query#iterate`, which is intended for loading entities when it is known that the loaded entries are already stored in the second-level cache. The idea behind iterate is that just the matching identifiers will be obtained in the SQL query. From these the identifiers are resolved by second-level cache lookup. If these second-level cache lookups fail, additional queries will need to be issued against the database.

> This operation can perform significantly better for loading large numbers of entities that for certain already exist in the second-level cache. In cases where many of the entities do not exist in the second-level cache, this operation will almost definitely perform worse.

The `Iterator` returned from `Query#iterate` is actually a specially typed Iterator: `org.hibernate.engine.HibernateIterator`. It is specialized to expose a `close()` method (again, inherited from `java.io.Closeable`). When you are done with this `Iterator` you should close it, either by casting to `HibernateIterator` or `Closeable`, or by calling `Hibernate#close(java.util.Iterator)` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Hibernate.html#close-java.util.Iterator-).

Since 5.2, Hibernate offers support for returning a `Stream` which can be later used to transform the underlying `ResultSet`.

Internally, the `stream()` behaves like a `Query#scroll` and the underlying result is backed by a `ScrollableResults`.

Fetching a projection using the `Query#stream` method can be done as follows:

Example 398. Hibernate `stream()` using a projection result type

```
try ( Stream<Object[]> persons = session.createQuery(
    "select p.name, p.nickName " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.stream() ) {

    persons
    .map( row -> new PersonNames(
            (String) row[0],
            (String) row[1] ) )
    .forEach( this::process );
}
```

JAVA

When fetching a single result, like a `Person` entity, instead of a `Stream<Object[]>`, Hibernate is going to figure out the actual type, so the result is a `Stream<Person>`.

Example 399. Hibernate `stream()` using an entity result type

```
try( Stream<Person> persons = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name" )
.setParameter( "name", "J%" )
.stream() ) {

    Map<Phone, List<Call>> callRegistry = persons
            .flatMap( person -> person.getPhones().stream() )
            .flatMap( phone -> phone.getCalls().stream() )
            .collect( Collectors.groupingBy( Call::getPhone ) );

    process(callRegistry);
}
```

JAVA

> Just like with `ScrollableResults`, you should always close a Hibernate `Stream` either explicitly or using a try-with-resources (https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) block.

15.5. Case Sensitivity

With the exception of names of Java classes and properties, queries are case-insensitive. So `SeLeCT` is the same as `sELEct` is the same as `SELECT`, but `org.hibernate.eg.FOO` and `org.hibernate.eg.Foo` are different, as are `foo.barSet` and `foo.BARSET`.

> This documentation uses lowercase keywords as convention in examples.

15.6. Statement types

Both HQL and JPQL allow `SELECT`, `UPDATE` and `DELETE` statements to be performed. HQL additionally allows `INSERT` statements, in a form similar to a SQL `INSERT FROM SELECT`.

> Care should be taken as to when a `UPDATE` or `DELETE` statement is executed.
>
> *Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a transaction in a new persistence con text or before fetching or accessing entities whose state might be affected by such operations.*
>
> *— Section 4.10 of the JPA 2.0 Specification*

15.7. Select statements

The BNF (https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form) for `SELECT` statements in HQL is:

```
select_statement :: =
    [select_clause]
    from_clause
    [where_clause]
    [groupby_clause]
    [having_clause]
    [orderby_clause]
```

SQL

The simplest possible HQL `SELECT` statement is of the form:

```
List<Person> persons = session.createQuery(
    "from Person" )
.list();
```

SQL

> The select statement in JPQL is exactly the same as for HQL except that JPQL requires a `select_clause`, whereas HQL does not.
>
> ```
> List<Person> persons = entityManager.createQuery(
>     "select p " +
>     "from Person p", Person.class )
> .getResultList();
> ```
>
> SQL
>
> Even though HQL does not require the presence of a `select_clause`, it is generally good practice to include one. For simple queries the intent is clear and so the intended result of the `select_clause` is east to infer. But on more complex queries that is not always the case.
>
> It is usually better to explicitly specify intent. Hibernate does not actually enforce that a `select_clause` be present even when parsing JPQL queries, however applications interested in JPA portability should take heed of this.

15.8. Update statements

The BNF for `UPDATE` statements is the same in HQL and JPQL:

```
update_statement ::=
    update_clause [where_clause]

update_clause ::=
    UPDATE entity_name [[AS] identification_variable]
    SET update_item {, update_item}*

update_item ::=
    [identification_variable.]{state_field | single_valued_object_field} = new_value

new_value ::=
    scalar_expression | simple_entity_expression | NULL
```
SQL

`UPDATE` statements, by default, do not effect the `version` or the `timestamp` attribute values for the affected entities.

However, you can force Hibernate to set the `version` or `timestamp` attribute values through the use of a `versioned update`. This is achieved by adding the `VERSIONED` keyword after the `UPDATE` keyword.

> This is a Hibernate specific feature and will not work in a portable manner.
>
> Custom version types, `org.hibernate.usertype.UserVersionType`, are not allowed in conjunction with a `update versioned` statement.

An `UPDATE` statement is executed using the `executeUpdate()` of either `org.hibernate.query.Query` or `javax.persistence.Query`. The method is named for those familiar with the JDBC `executeUpdate()` on `java.sql.PreparedStatement`.

The `int` value returned by the `executeUpdate()` method indicates the number of entities effected by the operation. This may or may not correlate to the number of rows effected in the database. An HQL bulk operation might result in multiple actual SQL statements being executed (for joined-subclass, for example). The returned number indicates the number of actual entities affected by the statement. Using a JOINED inheritance hierarchy, a delete against one of the subclasses may actually result in deletes against not just the table to which that subclass is mapped, but also the "root" table and tables "in between".

Example 400. UPDATE query statements

```
int updatedEntities = entityManager.createQuery(
    "update Person p " +
    "set p.name = :newName " +
    "where p.name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();

int updatedEntities = session.createQuery(
    "update Person " +
    "set name = :newName " +
    "where name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();

int updatedEntities = session.createQuery(
    "update versioned Person " +
    "set name = :newName " +
    "where name = :oldName" )
.setParameter( "oldName", oldName )
.setParameter( "newName", newName )
.executeUpdate();
```

SQL

> Neither `UPDATE` nor `DELETE` statements allow implicit joins. Their form already disallows explicit joins too.

15.9. Delete statements

The BNF for `DELETE` statements is the same in HQL and JPQL:

```
delete_statement ::=
    delete_clause [where_clause]

delete_clause ::=
    DELETE FROM entity_name [[AS] identification_variable]
```

SQL

A `DELETE` statement is also executed using the `executeUpdate()` method of either `org.hibernate.query.Query` or `javax.persistence.Query`.

15.10. Insert statements

HQL adds the ability to define `INSERT` statements as well.

> There is no JPQL equivalent to this.

The BNF for an HQL `INSERT` statement is:

```
SQL

insert_statement ::=
    insert_clause select_statement

insert_clause ::=
    INSERT INTO entity_name (attribute_list)

attribute_list ::=
    state_field[, state_field ]*
```

The `attribute_list` is analogous to the `column specification` in the SQL `INSERT` statement. For entities involved in mapped inheritance, only attributes directly defined on the named entity can be used in the `attribute_list`. Superclass properties are not allowed and subclass properties do not make sense. In other words, `INSERT` statements are inherently non-polymorphic.

`select_statement` can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. This may cause problems between Hibernate Types which are *equivalent* as opposed to *equal*. For example, this might cause lead to issues with mismatches between an attribute mapped as a `org.hibernate.type.DateType` and an attribute defined as a `org.hibernate.type.TimestampType`, even though the database might not make a distinction or might be able to handle the conversion.

For the id attribute, the insert statement gives you two options. You can either explicitly specify the id property in the `attribute_list`, in which case its value is taken from the corresponding select expression, or omit it from the `attribute_list` in which case a generated value is used. This latter option is only available when using id generators that operate "in the

database"; attempting to use this option with any "in memory" type generators will cause an exception during parsing.

For optimistic locking attributes, the insert statement again gives you two options. You can either specify the attribute in the `attribute_list` in which case its value is taken from the corresponding select expressions, or omit it from the `attribute_list` in which case the `seed value` defined by the corresponding `org.hibernate.type.VersionType` is used.

Example 401. INSERT query statements

```
int insertedEntities = session.createQuery(
    "insert into Partner (id, name) " +
    "select p.id, p.name " +
    "from Person p ")
.executeUpdate();
```

SQL

15.11. The `FROM` clause

The `FROM` clause is responsible defining the scope of object model types available to the rest of the query. It also is responsible for defining all the "identification variables" available to the rest of the query.

15.12. Identification variables

Identification variables are often referred to as aliases. References to object model classes in the `FROM` clause can be associated with an identification variable that can then be used to refer to that type throughout the rest of the query.

In most cases declaring an identification variable is optional, though it is usually good practice to declare them.

An identification variable must follow the rules for Java identifier validity.

According to JPQL, identification variables must be treated as case-insensitive. Good practice says you should use the same case throughout a query to refer to a given identification variable. In other words, JPQL says they *can be* case-insensitive and so Hibernate must be able to treat them as such, but this does not make it good practice.

15.13. Root entity references

A root entity reference, or what JPA calls a `range variable declaration`, is specifically a reference to a mapped entity type from the application. It cannot name component/ embeddable types. And associations, including collections, are handled in a different manner, as later discussed.

The BNF for a root entity reference is:

```
root_entity_reference ::=
    entity_name [AS] identification_variable
```

SQL

Example 402. Simple query example

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from org.hibernate.userguide.model.Person p", Person.class )
.getResultList();
```

JAVA

We see that the query is defining a root entity reference to the `org.hibernate.userguide.model.Person` object model type. Additionally, it declares an alias of `p` to that `org.hibernate.userguide.model.Person` reference, which is the identification variable.

Usually, the root entity reference represents just the `entity name` rather than the entity class FQN (fully-qualified name). By default, the entity name is the unqualified entity class name, here `Person`

Example 403. Simple query using entity name for root entity reference

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p", Person.class )
.getResultList();
```

JAVA

Multiple root entity references can also be specified, even when naming the same entity.

Example 404. Simple query using multiple root entity references

```
List<Object[]> persons = entityManager.createQuery(
    "select distinct pr, ph " +
    "from Person pr, Phone ph " +
    "where ph.person = pr and ph is not null", Object[].class)
.getResultList();
```

SQL

```
List<Person> persons = entityManager.createQuery(
    "select distinct pr1 " +
    "from Person pr1, Person pr2 " +
    "where pr1.id <> pr2.id " +
    "  and pr1.address = pr2.address " +
    "  and pr1.createdOn < pr2.createdOn", Person.class )
.getResultList();
```

JAVA

15.14. Explicit joins

The `FROM` clause can also contain explicit relationship joins using the `join` keyword. These joins can be either `inner` or `left outer` style joins.

Example 405. Explicit inner join examples

```
SQL
List<Person> persons = entityManager.createQuery(
    "select distinct pr " +
    "from Person pr " +
    "join pr.phones ph " +
    "where ph.type = :phoneType", Person.class )
.setParameter( "phoneType", PhoneType.MOBILE )
.getResultList();

// same query but specifying join type as 'inner' explicitly
List<Person> persons = entityManager.createQuery(
    "select distinct pr " +
    "from Person pr " +
    "inner join pr.phones ph " +
    "where ph.type = :phoneType", Person.class )
.setParameter( "phoneType", PhoneType.MOBILE )
.getResultList();
```

Example 406. Explicit left (outer) join examples

```
JAVA
List<Person> persons = entityManager.createQuery(
    "select distinct pr " +
    "from Person pr " +
    "left join pr.phones ph " +
    "where ph is null " +
    "   or ph.type = :phoneType", Person.class )
.setParameter( "phoneType", PhoneType.LAND_LINE )
.getResultList();

// functionally the same query but using the 'left outer' phrase
List<Person> persons = entityManager.createQuery(
    "select distinct pr " +
    "from Person pr " +
    "left outer join pr.phones ph " +
    "where ph is null " +
    "   or ph.type = :phoneType", Person.class )
.setParameter( "phoneType", PhoneType.LAND_LINE )
.getResultList();
```

HQL also defines a `WITH` clause to qualify the join conditions.

> This is specific to HQL. JPQL defines the `ON` clause for this feature.

Example 407. HQL `WITH` clause join example

```
List<Object[]> personsAndPhones = session.createQuery(
    "select pr.name, ph.number " +
    "from Person pr " +
    "left join pr.phones ph with ph.type = :phoneType " )
.setParameter( "phoneType", PhoneType.LAND_LINE )
.list();
```

JAVA

Example 408. JPQL `ON` clause join example

```
List<Object[]> personsAndPhones = entityManager.createQuery(
    "select pr.name, ph.number " +
    "from Person pr " +
    "left join pr.phones ph on ph.type = :phoneType " )
.setParameter( "phoneType", PhoneType.LAND_LINE )
.getResultList();
```

JAVA

> The important distinction is that in the generated SQL the conditions of the `WITH/ON` clause are made part of the `ON` clause in the generated SQL, as opposed to the other queries in this section where the HQL/JPQL conditions are made part of the `WHERE` clause in the generated SQL.

The distinction in this specific example is probably not that significant. The `with clause` is sometimes necessary for more complicated queries.

Explicit joins may reference association or component/embedded attributes. In the case of component/embedded attributes, the join is simply logical and does not correlate to a physical (SQL) join. For further information about collection-valued association references, see Collection member references.

An important use case for explicit joins is to define `FETCH JOINS` which override the laziness of the joined association. As an example, given an entity named `Person` with a collection-valued association named `phones`, the `JOIN FETCH` will also load the child collection in the same SQL query:

Example 409. Fetch join example

JAVA

```
// functionally the same query but using the 'left outer' phrase
List<Person> persons = entityManager.createQuery(
    "select distinct pr " +
    "from Person pr " +
    "left join fetch pr.phones ", Person.class )
.getResultList();
```

As you can see from the example, a fetch join is specified by injecting the keyword `fetch` after the keyword `join`. In the example, we used a left outer join because we also wanted to return customers who have no orders.

Inner joins can also be fetched, but inner joins filter out the root entity. In the example, using an inner join instead would have resulted in customers without any orders being filtered out of the result.

> Fetch joins are not valid in sub-queries.
>
> Care should be taken when fetch joining a collection-valued association which is in any way further restricted (the fetched collection will be restricted too). For this reason, it is usually considered best practice not to assign an identification variable to fetched joins except for the purpose of specifying nested fetch joins.
>
> Fetch joins should not be used in paged queries (e.g. `setFirstResult()` or `setMaxResults()`), nor should they be used with the `scroll()` or `iterate()` features.

15.15. Implicit joins (path expressions)

Another means of adding to the scope of object model types available to the query is through the use of implicit joins or path expressions.

Example 410. Simple implicit join example

JAVA

```
List<Phone> phones = entityManager.createQuery(
    "select ph " +
    "from Phone ph " +
    "where ph.person.address = :address ", Phone.class )
.setParameter( "address", address )
.getResultList();

// same as
List<Phone> phones = entityManager.createQuery(
    "select ph " +
    "from Phone ph " +
    "join ph.person pr " +
    "where pr.address = :address ", Phone.class )
.setParameter( "address", address)
.getResultList();
```

An implicit join always starts from an `identification variable`, followed by the navigation operator (`.`), followed by an attribute for the object model type referenced by the initial `identification variable`. In the example, the initial `identification variable` is `ph` which refers to the `Phone` entity. The `ph.person` reference then refers to the `person` attribute of the `Phone` entity. `person` is an association type so we further navigate to its age attribute.

> If the attribute represents an entity association (non-collection) or a component/embedded, that reference can be further navigated. Basic values and collection-valued associations cannot be further navigated.

As shown in the example, implicit joins can appear outside the `FROM clause`. However, they affect the `FROM clause`.

> Implicit joins are always treated as inner joins.
>
> Multiple references to the same implicit join always refer to the same logical and physical (SQL) join.

Example 411. Reused implicit join

JAVA

```
List<Phone> phones = entityManager.createQuery(
    "select ph " +
    "from Phone ph " +
    "where ph.person.address = :address " +
    "  and ph.person.createdOn > :timestamp", Phone.class )
.setParameter( "address", address )
.setParameter( "timestamp", timestamp )
.getResultList();

//same as
List<Phone> phones = entityManager.createQuery(
    "select ph " +
    "from Phone ph " +
    "inner join ph.person pr " +
    "where pr.address = :address " +
    "  and pr.createdOn > :timestamp", Phone.class )
.setParameter( "address", address )
.setParameter( "timestamp", timestamp )
.getResultList();
```

Just as with explicit joins, implicit joins may reference association or component/embedded attributes. For further information about collection-valued association references, see Collection member references.

In the case of component/embedded attributes, the join is simply logical and does not correlate to a physical (SQL) join. Unlike explicit joins, however, implicit joins may also reference basic state fields as long as the path expression ends there.

15.16. Distinct

For JPQL and HQL, `DISTINCT` has two meanings:

1. It can be passed to the database so that duplicates are removed from a result set
2. It can be used to filter out the same parent entity references when join fetching a child collection

15.16.1. Using DISTINCT with SQL projections

For SQL projections, `DISTINCT` needs to be passed to the database because the duplicated entries need to be filtered out before being returned to the database client.

Example 412. Using DISTINCT with projection queries example

```
List<String> lastNames = entityManager.createQuery(
    "select distinct p.lastName " +
    "from Person p", String.class)
.getResultList();
```
JAVA

When running the query above, Hibernate generates the following SQL query:

```
SELECT DISTINCT
    p.last_name as col_0_0_
FROM person p
```
SQL

For this particular use case, passing the `DISTINCT` keyword from JPQL/HQL to the database is the right thing to do.

15.16.2. Using DISTINCT with entity queries

`DISTINCT` can also be used to filter out entity object references when fetching a child association along with the parent entities.

Example 413. Using DISTINCT with entity queries example

```
List<Person> authors = entityManager.createQuery(
    "select distinct p " +
    "from Person p " +
    "left join fetch p.books", Person.class)
.getResultList();
```
JAVA

In this case, `DISTINCT` is used because there can be multiple `Books` entities associated to a given `Person`. If in the database there are 3 `Persons` in the database and each person has 2 `Books`, without `DISTINCT` this query will return 6 `Persons` since the SQL-level result-set size is given by the number of joined `Book` records.

However, the `DISTINCT` keyword is passed to the database as well:

```
SQL
SELECT DISTINCT
    p.id as id1_1_0_,
    b.id as id1_0_1_,
    p.first_name as first_na2_1_0_,
    p.last_name as last_nam3_1_0_,
    b.author_id as author_i3_0_1_,
    b.title as title2_0_1_,
    b.author_id as author_i3_0_0__,
    b.id as id1_0_0__
FROM person p
LEFT OUTER JOIN book b ON p.id=b.author_id
```

In this case, the `DISTINCT` SQL keyword is undesirable since it does a redundant result set sorting, as explained in this blog post (http://in.relation.to/2016/08/04/introducing-distinct-pass-through-query-hint/). To fix this issue, Hibernate 5.2.2 added support for the `HINT_PASS_DISTINCT_THROUGH` entity query hint:

Example 414. Using DISTINCT with entity queries example

```
JAVA
List<Person> authors = entityManager.createQuery(
    "select distinct p " +
    "from Person p " +
    "left join fetch p.books", Person.class)
.setHint( QueryHints.HINT_PASS_DISTINCT_THROUGH, false )
.getResultList();
```

With this entity query hint, Hibernate will not pass the `DISTINCT` keyword to the SQL query:

```
SQL
SELECT
    p.id as id1_1_0_,
    b.id as id1_0_1_,
    p.first_name as first_na2_1_0_,
    p.last_name as last_nam3_1_0_,
    b.author_id as author_i3_0_1_,
    b.title as title2_0_1_,
    b.author_id as author_i3_0_0__,
    b.id as id1_0_0__
FROM person p
LEFT OUTER JOIN book b ON p.id=b.author_id
```

When using the `HINT_PASS_DISTINCT_THROUGH` entity query hint, Hibernate can still remove the duplicated parent-side entities from the query result.

15.17. Collection member references

References to collection-valued associations actually refer to the *values* of that collection.

Example 415. Collection references example

```
List<Phone> phones = entityManager.createQuery(
    "select ph " +
    "from Person pr " +
    "join pr.phones ph " +
    "join ph.calls c " +
    "where pr.address = :address " +
    "  and c.duration > :duration", Phone.class )
.setParameter( "address", address )
.setParameter( "duration", duration )
.getResultList();

// alternate syntax
List<Phone> phones = session.createQuery(
    "select pr " +
    "from Person pr, " +
    "in (pr.phones) ph, " +
    "in (ph.calls) c " +
    "where pr.address = :address " +
    "  and c.duration > :duration" )
.setParameter( "address", address )
.setParameter( "duration", duration )
.list();
```

JAVA

In the example, the identification variable `ph` actually refers to the object model type `Phone`, which is the type of the elements of the `Person#phones` association.

The example also shows the alternate syntax for specifying collection association joins using the `IN` syntax. Both forms are equivalent. Which form an application chooses to use is simply a matter of taste.

15.18. Special case - qualified path expressions

We said earlier that collection-valued associations actually refer to the *values* of that collection. Based on the type of collection, there are also available a set of explicit qualification expressions.

Example 416. Qualified collection references example

JAVA

```
@OneToMany(mappedBy = "phone")
@MapKey(name = "timestamp")
@MapKeyTemporal(TemporalType.TIMESTAMP )
private Map<Date, Call> callHistory = new HashMap<>();


// select all the calls (the map value) for a given Phone
List<Call> calls = entityManager.createQuery(
    "select ch " +
    "from Phone ph " +
    "join ph.callHistory ch " +
    "where ph.id = :id ", Call.class )
.setParameter( "id", id )
.getResultList();

// same as above
List<Call> calls = entityManager.createQuery(
    "select value(ch) " +
    "from Phone ph " +
    "join ph.callHistory ch " +
    "where ph.id = :id ", Call.class )
.setParameter( "id", id )
.getResultList();

// select all the Call timestamps (the map key) for a given Phone
List<Date> timestamps = entityManager.createQuery(
    "select key(ch) " +
    "from Phone ph " +
    "join ph.callHistory ch " +
    "where ph.id = :id ", Date.class )
.setParameter( "id", id )
.getResultList();

// select all the Call and their timestamps (the 'Map.Entry') for a given Phone
List<Map.Entry<Date, Call>> callHistory = entityManager.createQuery(
    "select entry(ch) " +
    "from Phone ph " +
    "join ph.callHistory ch " +
    "where ph.id = :id " )
.setParameter( "id", id )
.getResultList();

// Sum all call durations for a given Phone of a specific Person
Long duration = entityManager.createQuery(
    "select sum(ch.duration) " +
    "from Person pr " +
    "join pr.phones ph " +
    "join ph.callHistory ch " +
    "where ph.id = :id " +
    "  and index(ph) = :phoneIndex", Long.class )
.setParameter( "id", id )
.setParameter( "phoneIndex", phoneIndex )
.getSingleResult();
```

VALUE

Refers to the collection value. Same as not specifying a qualifier. Useful to explicitly show intent. Valid for any type of collection-valued reference.

INDEX

According to HQL rules, this is valid for both `Maps` and `Lists` which specify a `javax.persistence.OrderColumn` annotation to refer to the `Map` key or the `List` position (aka the `OrderColumn` value). JPQL however, reserves this for use in the `List` case and adds `KEY` for the `Map` case. Applications interested in JPA provider portability should be aware of this distinction.

KEY

Valid only for `Maps` . Refers to the map's key. If the key is itself an entity, it can be further navigated.

ENTRY

Only valid for `Maps` . Refers to the map's logical `java.util.Map.Entry` tuple (the combination of its key and value). `ENTRY` is only valid as a terminal path and it's applicable to the `SELECT` clause only.

See Collection-related expressions for additional details on collection related expressions.

15.19. Polymorphism

HQL and JPQL queries are inherently polymorphic.

JAVA

```
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p ", Payment.class )
.getResultList();
```

This query names the `Payment` entity explicitly. However, all subclasses of `Payment` are also available to the query. So if the `CreditCardPayment` and `WireTransferPayment` entities extend the `Payment` class, all three types would be available to the entity query, and the query would return instances of all three.

> This can be altered by using either the `org.hibernate.annotations.Polymorphism` annotation (global, and Hibernate-specific) or limiting them using in the query itself using an entity type expression.
>
> The HQL query `from java.lang.Object` is totally valid! It returns every object of every type defined in your application.

15.20. Expressions

Essentially expressions are references that resolve to basic or tuple values.

15.21. Identification variable

See The `FROM` clause.

15.22. Path expressions

Again, see The `FROM` clause.

15.23. Literals

String literals are enclosed in single quotes. To escape a single quote within a string literal, use double single quotes.

Example 417. String literals examples

```
JAVA
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like 'Joe'", Person.class)
.getResultList();

// Escaping quotes
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like 'Joe''s'", Person.class)
.getResultList();
```

Numeric literals are allowed in a few different forms.

Example 418. Numeric literal examples

JAVA

```
// simple integer literal
Person person = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.id = 1", Person.class)
.getSingleResult();

// simple integer literal, typed as a long
Person person = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.id = 1L", Person.class)
.getSingleResult();

// decimal notation
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration > 100.5", Call.class )
.getResultList();

// decimal notation, typed as a float
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration > 100.5F", Call.class )
.getResultList();

// scientific notation
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration > 1e+2", Call.class )
.getResultList();

// scientific notation, typed as a float
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration > 1e+2F", Call.class )
.getResultList();
```

In the scientific notation form, the `E` is case-insensitive.

Specific typing can be achieved through the use of the same suffix approach specified by Java. So, `L` denotes a long, `D` denotes a double, `F` denotes a float. The actual suffix is case-insensitive.

The boolean literals are `TRUE` and `FALSE`, again case-insensitive.

Enums can even be referenced as literals. The fully-qualified enum class name must be used. HQL can also handle constants in the same manner, though JPQL does not define that as supported.

Entity names can also be used as literal. See Entity type.

Date/time literals can be specified using the JDBC escape syntax:

- `{d 'yyyy-mm-dd'}` for dates
- `{t 'hh:mm:ss'}` for times
- `{ts 'yyyy-mm-dd hh:mm:ss[.millis]'}` (millis optional) for timestamps.

These Date/time literals only work if you JDBC drivers supports them.

15.24. Arithmetic

Arithmetic operations also represent valid expressions.

Example 419. Numeric arithmetic examples

```java
// select clause date/time arithmetic operations
Long duration = entityManager.createQuery(
    "select sum(ch.duration) * :multiplier " +
    "from Person pr " +
    "join pr.phones ph " +
    "join ph.callHistory ch " +
    "where ph.id = 1L ", Long.class )
.setParameter( "multiplier", 1000L )
.getSingleResult();

// select clause date/time arithmetic operations
Integer years = entityManager.createQuery(
    "select year( current_date() ) - year( p.createdOn ) " +
    "from Person p " +
    "where p.id = 1L", Integer.class )
.getSingleResult();

// where clause arithmetic operations
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where year( current_date() ) - year( p.createdOn ) > 1", Person.class )
.getResultList();
```

The following rules apply to the result of arithmetic operations:

- If either of the operands is `Double` / `double` , the result is a `Double`
- else, if either of the operands is `Float` / `float` , the result is a `Float`
- else, if either operand is `BigDecimal` , the result is `BigDecimal`
- else, if either operand is `BigInteger` , the result is `BigInteger` (except for division, in which case the result type is not further defined)
- else, if either operand is `Long` / `long` , the result is `Long` (except for division, in which case the result type is not further defined)
- else, (the assumption being that both operands are of integral type) the result is `Integer` (except for division, in which case the result type is not further defined)

Date arithmetic is also supported, albeit in a more limited fashion. This is due partially to differences in database support and partially to the lack of support for `INTERVAL` definition in the query language itself.

15.25. Concatenation (operation)

HQL defines a concatenation operator in addition to supporting the concatenation (`CONCAT`) function. This is not defined by JPQL, so portable applications should avoid it use. The concatenation operator is taken from the SQL concatenation operator (e.g `||`).

Example 420. Concatenation operation example

```
String name = entityManager.createQuery(
    "select 'Customer ' || p.name " +
    "from Person p " +
    "where p.id = 1", String.class )
.getSingleResult();
```

JAVA

See Scalar functions for details on the `concat()` function

15.26. Aggregate functions

Aggregate functions are also valid expressions in HQL and JPQL. The semantic is the same as their SQL counterpart. The supported aggregate functions are:

`COUNT` (including distinct/all qualifiers)
: The result type is always `Long`.

`AVG`
: The result type is always `Double`.

`MIN`
: The result type is the same as the argument type.

`MAX`
: The result type is the same as the argument type.

`SUM`
: The result type of the `SUM()` function depends on the type of the values being summed. For integral values (other than `BigInteger`), the result type is `Long`.

For floating point values (other than `BigDecimal`) the result type is `Double`. For `BigInteger` values, the result type is `BigInteger`. For `BigDecimal` values, the result type is `BigDecimal`.

Example 421. Aggregate function examples

```
Object[] callStatistics = entityManager.createQuery(
    "select " +
    "    count(c), " +
    "    sum(c.duration), " +
    "    min(c.duration), " +
    "    max(c.duration), " +
    "    avg(c.duration)  " +
    "from Call c ", Object[].class )
.getSingleResult();

Long phoneCount = entityManager.createQuery(
    "select count( distinct c.phone ) " +
    "from Call c ", Long.class )
.getSingleResult();

List<Object[]> callCount = entityManager.createQuery(
    "select p.number, count(c) " +
    "from Call c " +
    "join c.phone p " +
    "group by p.number", Object[].class )
.getResultList();
```
JAVA

Aggregations often appear with grouping. For information on grouping see Group by.

15.27. Scalar functions

Both HQL and JPQL define some standard functions that are available regardless of the underlying database in use. HQL can also understand additional functions defined by the Dialect as well as the application.

15.28. JPQL standardized functions

Here is the list of functions defined as supported by JPQL. Applications interested in remaining portable between JPA providers should stick to these functions.

CONCAT

String concatenation function. Variable argument length of 2 or more string values to be concatenated together.

```
List<String> callHistory = entityManager.createQuery(
    "select concat( p.number, ' : ' ,c.duration ) " +
    "from Call c " +
    "join c.phone p", String.class )
.getResultList();
```
JAVA

SUBSTRING

Extracts a portion of a string value. The second argument denotes the starting position, where 1 is the first character of the string. The third (optional) argument denotes the length.

```
List<String> prefixes = entityManager.createQuery(
    "select substring( p.number, 1, 2 ) " +
    "from Call c " +
    "join c.phone p", String.class )
.getResultList();
```
JAVA

UPPER

Upper cases the specified string

```
List<String> names = entityManager.createQuery(
    "select upper( p.name ) " +
    "from Person p ", String.class )
.getResultList();
```
JAVA

LOWER

Lower cases the specified string

```
List<String> names = entityManager.createQuery(
    "select lower( p.name ) " +
    "from Person p ", String.class )
.getResultList();
```
JAVA

TRIM

Follows the semantics of the SQL trim function.

```
List<String> names = entityManager.createQuery(
    "select trim( p.name ) " +
    "from Person p ", String.class )
.getResultList();
```
JAVA

LENGTH

Returns the length of a string.

```java
List<Integer> lengths = entityManager.createQuery(
    "select length( p.name ) " +
    "from Person p ", Integer.class )
.getResultList();
```

LOCATE

Locates a string within another string. The third argument (optional) is used to denote a position from which to start looking.

```java
List<Integer> sizes = entityManager.createQuery(
    "select locate( 'John', p.name ) " +
    "from Person p ", Integer.class )
.getResultList();
```

ABS

Calculates the mathematical absolute value of a numeric value.

```java
List<Integer> abs = entityManager.createQuery(
    "select abs( c.duration ) " +
    "from Call c ", Integer.class )
.getResultList();
```

MOD

Calculates the remainder of dividing the first argument by the second.

```java
List<Integer> mods = entityManager.createQuery(
    "select mod( c.duration, 10 ) " +
    "from Call c ", Integer.class )
.getResultList();
```

SQRT

Calculates the mathematical square root of a numeric value.

```
List<Double> sqrts = entityManager.createQuery(
    "select sqrt( c.duration ) " +
    "from Call c ", Double.class )
.getResultList();
```
JAVA

CURRENT_DATE

Returns the database current date.

```
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.timestamp = current_date", Call.class )
.getResultList();
```
JAVA

CURRENT_TIME

Returns the database current time.

```
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.timestamp = current_time", Call.class )
.getResultList();
```
JAVA

CURRENT_TIMESTAMP

Returns the database current timestamp.

```
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.timestamp = current_timestamp", Call.class )
.getResultList();
```
JAVA

15.29. HQL functions

Beyond the JPQL standardized functions, HQL makes some additional functions available regardless of the underlying database in use.

BIT_LENGTH

Returns the length of binary data.

```
List<Number> bits = entityManager.createQuery(
    "select bit_length( c.duration ) " +
    "from Call c ", Number.class )
.getResultList();
```
JAVA

CAST

Performs a SQL cast. The cast target should name the Hibernate mapping type to use. See the data types chapter on for more information.

```
List<String> durations = entityManager.createQuery(
    "select cast( c.duration as string ) " +
    "from Call c ", String.class )
.getResultList();
```
JAVA

EXTRACT

Performs a SQL extraction on datetime values. An extraction extracts parts of the datetime (the year, for example).

```
List<Integer> years = entityManager.createQuery(
    "select extract( YEAR from c.timestamp ) " +
    "from Call c ", Integer.class )
.getResultList();
```
JAVA

See the abbreviated forms below.

YEAR

Abbreviated extract form for extracting the year.

```
List<Integer> years = entityManager.createQuery(
    "select year( c.timestamp ) " +
    "from Call c ", Integer.class )
.getResultList();
```
JAVA

MONTH

Abbreviated extract form for extracting the month.

DAY

Abbreviated extract form for extracting the day.

HOUR

Abbreviated extract form for extracting the hour.

MINUTE

Abbreviated extract form for extracting the minute.

SECOND

Abbreviated extract form for extracting the second.

STR

Abbreviated form for casting a value as character data.

```
List<String> timestamps = entityManager.createQuery(
    "select str( c.timestamp ) " +
    "from Call c ", String.class )
.getResultList();
```

JAVA

15.30. Non-standardized functions

Hibernate Dialects can register additional functions known to be available for that particular database product. These functions are also available in HQL (and JPQL, though only when using Hibernate as the JPA provider obviously). However, they would only be available when using that database Dialect. Applications that aim for database portability should avoid using functions in this category.

Application developers can also supply their own set of functions. This would usually represent either custom SQL functions or aliases for snippets of SQL. Such function declarations are made by using the `addSqlFunction()` method of `org.hibernate.cfg.Configuration`.

15.31. Collection-related expressions

There are a few specialized expressions for working with collection-valued associations. Generally, these are just abbreviated forms or other expressions for the sake of conciseness.

SIZE

Calculate the size of a collection. Equates to a subquery!

MAXELEMENT

Available for use on collections of basic type. Refers to the maximum value as determined by applying the `max` SQL aggregation.

MAXINDEX

Available for use on indexed collections. Refers to the maximum index (key/position) as determined by applying the `max` SQL aggregation.

MINELEMENT

Available for use on collections of basic type. Refers to the minimum value as determined by applying the `min` SQL aggregation.

MININDEX

Available for use on indexed collections. Refers to the minimum index (key/position) as determined by applying the `min` SQL aggregation.

ELEMENTS

Used to refer to the elements of a collection as a whole. Only allowed in the where clause. Often used in conjunction with `ALL`, `ANY` or `SOME` restrictions.

INDICES

Similar to `elements` except that `indices` refers to the collections indices (keys/positions) as a whole.

Example 422. Collection-related expressions examples

JAVA

```
List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where maxelement( p.calls ) = :call", Phone.class )
.setParameter( "call", call )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where minelement( p.calls ) = :call", Phone.class )
.setParameter( "call", call )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where maxindex( p.phones ) = 0", Person.class )
.getResultList();

// the above query can be re-written with member of
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where :phone member of p.phones", Person.class )
.setParameter( "phone", phone )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where :phone = some elements ( p.phones )", Person.class )
.setParameter( "phone", phone )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where exists elements ( p.phones )", Person.class )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where current_date() > key( p.callHistory )", Phone.class )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where current_date() > all elements( p.repairTimestamps )", Phone.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
```

```
    "where 1 in indices( p.phones )", Person.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where size( p.phones ) = 2", Person.class )
.getResultList();
```

Elements of indexed collections (arrays, lists, and maps) can be referred to by index operator.

Example 423. Index operator examples

JAVA

```
// indexed lists
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.phones[ 0 ].type = 'LAND_LINE'", Person.class )
.getResultList();

// maps
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.addresses[ 'HOME' ] = :address", Person.class )
.setParameter( "address", address)
.getResultList();

//max index in list
List<Person> persons = entityManager.createQuery(
    "select pr " +
    "from Person pr " +
    "where pr.phones[ maxindex(pr.phones) ].type = 'LAND_LINE'", Person.class )
.getResultList();
```

See also Special case - qualified path expressions as there is a good deal of overlap.

15.32. Entity type

We can also refer to the type of an entity as an expression. This is mainly useful when dealing with entity inheritance hierarchies. The type can expressed using a `TYPE` function used to refer to the type of an identification variable representing an entity. The name of the entity also serves as a way to refer to an entity type. Additionally, the entity type can be parameterized, in which case the entity's Java Class reference would be bound as the parameter value.

Example 424. Entity type expression examples

```
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p " +
    "where type(p) = CreditCardPayment", Payment.class )
.getResultList();
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p " +
    "where type(p) = :type", Payment.class )
.setParameter( "type", WireTransferPayment.class)
.getResultList();
```
JAVA

> HQL also has a legacy form of referring to an entity type, though that legacy form is considered deprecated in favor of `TYPE`. The legacy form would have used `p.class` in the examples rather than `type(p)`. It is mentioned only for completeness.

15.33. CASE expressions

Both the simple and searched forms are supported, as well as the two SQL defined abbreviated forms (`NULLIF` and `COALESCE`)

15.34. Simple CASE expressions

The simple form has the following syntax:

```
CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} END
```
JAVA

Example 425. Simple case expression example

```
List<String> nickNames = entityManager.createQuery(
    "select " +
    "    case p.nickName " +
    "    when 'NA' " +
    "    then '<no nick name>' " +
    "    else p.nickName " +
    "    end " +
    "from Person p", String.class )
.getResultList();

// same as above
List<String> nickNames = entityManager.createQuery(
    "select coalesce(p.nickName, '<no nick name>') " +
    "from Person p", String.class )
.getResultList();
```
JAVA

15.35. Searched CASE expressions

The searched form has the following syntax:

```
CASE [ WHEN {test_conditional} THEN {match_result} ]* ELSE {miss_result} END
```
JAVA

Example 426. Searched case expression example

```
List<String> nickNames = entityManager.createQuery(
    "select " +
    "    case " +
    "    when p.nickName is null " +
    "    then " +
    "        case " +
    "        when p.name is null " +
    "        then '<no nick name>' " +
    "        else p.name " +
    "        end" +
    "    else p.nickName " +
    "    end " +
    "from Person p", String.class )
.getResultList();

// coalesce can handle this more succinctly
List<String> nickNames = entityManager.createQuery(
    "select coalesce( p.nickName, p.name, '<no nick name>' ) " +
    "from Person p", String.class )
.getResultList();
```
JAVA

15.36. NULLIF expressions

NULLIF is an abbreviated CASE expression that returns NULL if its operands are considered equal.

Example 427. NULLIF example

```java
List<String> nickNames = entityManager.createQuery(
    "select nullif( p.nickName, p.name ) " +
    "from Person p", String.class )
.getResultList();

// equivalent CASE expression
List<String> nickNames = entityManager.createQuery(
    "select " +
    "    case" +
    "    when p.nickName = p.name" +
    "    then null" +
    "    else p.nickName" +
    "    end " +
    "from Person p", String.class )
.getResultList();
```

15.37. COALESCE expressions

`COALESCE` is an abbreviated CASE expression that returns the first non-null operand. We have seen a number of `COALESCE` examples above.

15.38. The `SELECT` clause

The `SELECT` clause identifies which objects and values to return as the query results. The expressions discussed in Expressions are all valid select expressions, except where otherwise noted. See the section Hibernate Query API for information on handling the results depending on the types of values specified in the `SELECT` clause.

There is a particular expression type that is only valid in the select clause. Hibernate calls this "dynamic instantiation". JPQL supports some of that feature and calls it a "constructor expression".

So rather than dealing with the `Object[]` (again, see Hibernate Query API) here we are wrapping the values in a type-safe java object that will be returned as the results of the query.

Example 428. Dynamic HQL and JPQL instantiation example

JAVA

```
public class CallStatistics {

    private final long count;
    private final long total;
    private final int min;
    private final int max;
    private final double abg;

    public CallStatistics(long count, long total, int min, int max, double abg) {
        this.count = count;
        this.total = total;
        this.min = min;
        this.max = max;
        this.abg = abg;
    }

    //Getters and setters omitted for brevity
}

CallStatistics callStatistics = entityManager.createQuery(
    "select new org.hibernate.userguide.hql.CallStatistics(" +
    "    count(c), " +
    "    sum(c.duration), " +
    "    min(c.duration), " +
    "    max(c.duration), " +
    "    avg(c.duration)" +
    ")  " +
    "from Call c ", CallStatistics.class )
.getSingleResult();
```

> The class reference must be fully qualified and it must have a matching constructor.

> The class here need not be mapped. If it does represent an entity, the resulting instances are returned in the NEW state (not managed!).

HQL supports additional "dynamic instantiation" features. First, the query can specify to return a `List` rather than an `Object[]` for scalar results:

Example 429. Dynamic instantiation example - list

```
List<List> phoneCallDurations = entityManager.createQuery(
    "select new list(" +
    "    p.number, " +
    "    c.duration " +
    ")  " +
    "from Call c " +
    "join c.phone p ", List.class )
.getResultList();
```

JAVA

The results from this query will be a `List<List>` as opposed to a `List<Object[]>`

HQL also supports wrapping the scalar results in a `Map`.

Example 430. Dynamic instantiation example - map

```
List<Map> phoneCallTotalDurations = entityManager.createQuery(
    "select new map(" +
    "    p.number as phoneNumber , " +
    "    sum(c.duration) as totalDuration, " +
    "    avg(c.duration) as averageDuration " +
    ")  " +
    "from Call c " +
    "join c.phone p " +
    "group by p.number ", Map.class )
.getResultList();
```

JAVA

The results from this query will be a `List<Map<String, Object>>` as opposed to a `List<Object[]>`. The keys of the map are defined by the aliases given to the select expressions. If the user doesn't assign aliases, the key will be the index of each particular result set column (e.g. 0, 1, 2, etc).

15.39. Predicates

Predicates form the basis of the where clause, the having clause and searched case expressions. They are expressions which resolve to a truth value, generally `TRUE` or `FALSE`, although boolean comparisons involving `NULL` generally resolve to `UNKNOWN`.

15.40. Relational comparisons

Comparisons involve one of the comparison operators: `=`, `>`, `>=`, `<`, `<=`, `<>`. HQL also defines `!=` as a comparison operator synonymous with `<>`. The operands should be of the same type.

Example 431. Relational comparison examples

JAVA

```
// numeric comparison
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration < 30 ", Call.class )
.getResultList();

// string comparison
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like 'John%' ", Person.class )
.getResultList();

// datetime comparison
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.createdOn > '1950-01-01' ", Person.class )
.getResultList();

// enum comparison
List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where p.type = 'MOBILE' ", Phone.class )
.getResultList();

// boolean comparison
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p " +
    "where p.completed = true ", Payment.class )
.getResultList();

// boolean comparison
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p " +
    "where type(p) = WireTransferPayment ", Payment.class )
.getResultList();

// entity value comparison
List<Object[]> phonePayments = entityManager.createQuery(
    "select p " +
    "from Payment p, Phone ph " +
    "where p.person = ph.person ", Object[].class )
.getResultList();
```

Comparisons can also involve subquery qualifiers: `ALL`, `ANY`, `SOME`. `SOME` and `ANY` are synonymous.

The `ALL` qualifier resolves to true if the comparison is true for all of the values in the result of the subquery. It resolves to false if the subquery result is empty.

Example 432. ALL subquery comparison qualifier example

```
// select all persons with all calls shorter than 50 seconds
List<Person> persons = entityManager.createQuery(
    "select distinct p.person " +
    "from Phone p " +
    "join p.calls c " +
    "where 50 > all ( " +
    "    select duration" +
    "    from Call" +
    "    where phone = p " +
    ") ", Person.class )
.getResultList();
```

JAVA

The `ANY` / `SOME` qualifier resolves to true if the comparison is true for some of (at least one of) the values in the result of the subquery. It resolves to false if the subquery result is empty.

15.41. Nullness predicate

Check a value for nullness. Can be applied to basic attribute references, entity references and parameters. HQL additionally allows it to be applied to component/embeddable types.

Example 433. Nullness checking examples

```
// select all persons with a nickname
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.nickName is not null", Person.class )
.getResultList();

// select all persons without a nickname
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.nickName is null", Person.class )
.getResultList();
```

JAVA

15.42. Like predicate

Performs a like comparison on string values. The syntax is:

```
like_expression ::=
    string_expression
    [NOT] LIKE pattern_value
    [ESCAPE escape_character]
```

JAVA

The semantics follow that of the SQL like expression. The `pattern_value` is the pattern to attempt to match in the `string_expression`. Just like SQL, `pattern_value` can use `_` and `%` as wildcards. The meanings are the same. The `_` symbol matches any single character and `%` matches any number of characters.

Example 434. Like predicate examples

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like 'Jo%'", Person.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name not like 'Jo%'", Person.class )
.getResultList();
```

JAVA

The optional `escape 'escape character'` is used to specify an escape character used to escape the special meaning of `_` and `%` in the `pattern_value`. This is useful when needing to search on patterns including either `_` or `%`.

The syntax is formed as follows: `'like_predicate' escape 'escape_symbol'` So, if `|` is the escape symbol and we want to match all stored procedures prefixed with `Dr_`, the like criteria becomes: `'Dr|_%' escape '|'`:

Example 435. Like with escape symbol

```
// find any person with a name starting with "Dr_"
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like 'Dr|_%' escape '|'", Person.class )
.getResultList();
```

JAVA

15.43. Between predicate

Analogous to the SQL `BETWEEN` expression, it checks if the value is within boundaries. All the operands should have comparable types.

Example 436. Between predicate examples

JAVA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "join p.phones ph " +
    "where p.id = 1L and index(ph) between 0 and 3", Person.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.createdOn between '1999-01-01' and '2001-01-02'", Person.class )
.getResultList();

List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "where c.duration between 5 and 20", Call.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name between 'H' and 'M'", Person.class )
.getResultList();
```

15.44. In predicate

`IN` predicates performs a check that a particular value is in a list of values. Its syntax is:

JAVA

```
in_expression ::=
    single_valued_expression [NOT] IN single_valued_list

single_valued_list ::=
    constructor_expression | (subquery) | collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

The types of the `single_valued_expression` and the individual values in the `single_valued_list` must be consistent.

JPQL limits the valid types here to string, numeric, date, time, timestamp, and enum types, and , in JPQL, `single_valued_expression` can only refer to:

- "state fields", which is its term for simple attributes. Specifically this excludes association and component/embedded attributes.
- entity type expressions. See Entity type

In HQL, `single_valued_expression` can refer to a far more broad set of expression types. Single-valued association are allowed, and so are component/embedded attributes, although that feature depends on the level of support for tuple or "row value constructor syntax" in the underlying database. Additionally, HQL does not limit the value type in any way, though application developers should be aware that different types may incur limited support based on the underlying database vendor. This is largely the reason for the JPQL limitations.

The list of values can come from a number of different sources. In the `constructor_expression` and `collection_valued_input_parameter`, the list of values must not be empty; it must contain at least one value.

Example 437. In predicate examples

JAVA

```
List<Payment> payments = entityManager.createQuery(
    "select p " +
    "from Payment p " +
    "where type(p) in ( CreditCardPayment, WireTransferPayment )", Payment.class )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where type in ( 'MOBILE', 'LAND_LINE' )", Phone.class )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select p " +
    "from Phone p " +
    "where type in :types", Phone.class )
.setParameter( "types", Arrays.asList( PhoneType.MOBILE, PhoneType.LAND_LINE ) )
.getResultList();

List<Phone> phones = entityManager.createQuery(
    "select distinct p " +
    "from Phone p " +
    "where p.person.id in (" +
    "    select py.person.id " +
    "    from Payment py" +
    "    where py.completed = true and py.amount > 50 " +
    ")", Phone.class )
.getResultList();

// Not JPQL compliant!
List<Phone> phones = entityManager.createQuery(
    "select distinct p " +
    "from Phone p " +
    "where p.person in (" +
    "    select py.person " +
    "    from Payment py" +
    "    where py.completed = true and py.amount > 50 " +
    ")", Phone.class )
.getResultList();

// Not JPQL compliant!
List<Payment> payments = entityManager.createQuery(
    "select distinct p " +
    "from Payment p " +
    "where ( p.amount, p.completed ) in (" +
    "    (50, true )," +
    "    (100, true )," +
    "    (5, false )" +
    ")", Payment.class )
.getResultList();
```

15.45. Exists predicate

Exists expressions test the existence of results from a subquery. The affirmative form returns true if the subquery result contains values. The negated form returns true if the subquery result is empty.

15.46. Empty collection predicate

The `IS [NOT] EMPTY` expression applies to collection-valued path expressions. It checks whether the particular collection has any associated values.

Example 438. Empty collection expression examples

JAVA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.phones is empty", Person.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.phones is not empty", Person.class )
.getResultList();
```

15.47. Member-of collection predicate

The `[NOT] MEMBER [OF]` expression applies to collection-valued path expressions. It checks whether a value is a member of the specified collection.

Example 439. Member-of collection expression examples

JAVA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where 'Home address' member of p.addresses", Person.class )
.getResultList();

List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where 'Home address' not member of p.addresses", Person.class )
.getResultList();
```

15.48. NOT predicate operator

The `NOT` operator is used to negate the predicate that follows it. If that following predicate is true, the NOT resolves to false.

If the predicate is true, NOT resolves to false. If the predicate is unknown (e.g. `NULL`), the NOT resolves to unknown as well.

15.49. AND predicate operator

The `AND` operator is used to combine 2 predicate expressions. The result of the AND expression is true if and only if both predicates resolve to true. If either predicates resolves to unknown, the AND expression resolves to unknown as well. Otherwise, the result is false.

15.50. OR predicate operator

The `OR` operator is used to combine 2 predicate expressions. The result of the OR expression is true if one predicate resolves to true. If both predicates resolve to unknown, the OR expression resolves to unknown. Otherwise, the result is false.

15.51. The `WHERE` clause

The `WHERE` clause of a query is made up of predicates which assert whether values in each potential row match the current filtering criteria. Thus, the where clause restricts the results returned from a select query and limits the scope of update and delete queries.

15.52. Group by

The `GROUP BY` clause allows building aggregated results for various value groups. As an example, consider the following queries:

Example 440. Group by example

```java
Long totalDuration = entityManager.createQuery(
    "select sum( c.duration ) " +
    "from Call c ", Long.class )
.getSingleResult();

List<Object[]> personTotalCallDurations = entityManager.createQuery(
    "select p.name, sum( c.duration ) " +
    "from Call c " +
    "join c.phone ph " +
    "join ph.person p " +
    "group by p.name", Object[].class )
.getResultList();

//It's even possible to group by entities!
List<Object[]> personTotalCallDurations = entityManager.createQuery(
    "select p, sum( c.duration ) " +
    "from Call c " +
    "join c.phone ph " +
    "join ph.person p " +
    "group by p", Object[].class )
.getResultList();
```

The first query retrieves the complete total of all orders. The second retrieves the total for each customer, grouped by each customer.

In a grouped query, the where clause applies to the non-aggregated values (essentially it determines whether rows will make it into the aggregation). The `HAVING` clause also restricts results, but it operates on the aggregated values. In the Group by example, we retrieved `Call` duration totals for all persons. If that ended up being too much data to deal with, we might want to restrict the results to focus only on customers with a summed total of more than 1000:

Example 441. Having example

```java
List<Object[]> personTotalCallDurations = entityManager.createQuery(
    "select p.name, sum( c.duration ) " +
    "from Call c " +
    "join c.phone ph " +
    "join ph.person p " +
    "group by p.name " +
    "having sum( c.duration ) > 1000", Object[].class )
.getResultList();
```

The `HAVING` clause follows the same rules as the `WHERE` clause and is also made up of predicates. `HAVING` is applied after the groupings and aggregations have been done, while the `WHERE` clause is applied before.

15.53. Order by

The results of the query can also be ordered. The `ORDER BY` clause is used to specify the selected values to be used to order the result. The types of expressions considered valid as part of the `ORDER BY` clause include:

- state fields
- component/embeddable attributes
- scalar expressions such as arithmetic operations, functions, etc.
- identification variable declared in the select clause for any of the previous expression types

Additionally, JPQL says that all values referenced in the `ORDER BY` clause must be named in the `SELECT` clause. HQL does not mandate that restriction, but applications desiring database portability should be aware that not all databases support referencing values in the `ORDER BY` clause that are not referenced in the select clause.

Individual expressions in the order-by can be qualified with either `ASC` (ascending) or `DESC` (descending) to indicated the desired ordering direction. Null values can be placed in front or at the end of the sorted set using `NULLS FIRST` or `NULLS LAST` clause respectively.

Example 442. Order by example

JAVA

```
List<Person> persons = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "order by p.name", Person.class )
.getResultList();

List<Object[]> personTotalCallDurations = entityManager.createQuery(
    "select p.name, sum( c.duration ) as total " +
    "from Call c " +
    "join c.phone ph " +
    "join ph.person p " +
    "group by p.name " +
    "order by total", Object[].class )
.getResultList();
```

15.54. Read-only entities

As explained in entity immutability section, fetching entities in read-only mode is much more efficient than fetching read-write entities. Even if the entities are mutable, you can still fetch them in read-only mode, and benefit from reducing the memory footprint and sepeding up the flushing process.

Read-only entities are skipped by the dirty checking mechanism as illustrated by the following example:

Example 443. Read-only entities query example

JAVA

```
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "join c.phone p " +
    "where p.number = :phoneNumber ", Call.class )
.setParameter( "phoneNumber", "123-456-7890" )
.setHint( "org.hibernate.readOnly", true )
.getResultList();

calls.forEach( c -> c.setDuration( 0 ) );
```

SQL

```
SELECT c.id AS id1_5_ ,
       c.duration AS duration2_5_ ,
       c.phone_id AS phone_id4_5_ ,
       c.call_timestamp AS call_tim3_5_
FROM   phone_call c
INNER JOIN phone p ON c.phone_id = p.id
WHERE   p.phone_number = '123-456-7890'
```

As you can see, there is no SQL `UPDATE` being executed.

The Hibernate native API offers a `Query#setReadOnly` method, as an alternative to using a JPA query hint:

Example 444. Read-only entities native query example

JAVA

```
List<Call> calls = entityManager.createQuery(
    "select c " +
    "from Call c " +
    "join c.phone p " +
    "where p.number = :phoneNumber ", Call.class )
.setParameter( "phoneNumber", "123-456-7890" )
.unwrap( org.hibernate.query.Query.class )
.setReadOnly( true )
.getResultList();
```

16. Criteria

Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.

> Hibernate offers an older, legacy `org.hibernate.Criteria` API which should be considered deprecated. No feature development will target those APIs. Eventually, Hibernate-specific criteria features will be ported as extensions to the JPA `javax.persistence.criteria.CriteriaQuery`. For details on the `org.hibernate.Criteria` API, see Legacy Hibernate Criteria Queries.

This chapter will focus on the JPA APIs for declaring type-safe criteria queries.

Criteria queries are a programmatic, type-safe way to express a query. They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, the select clause, or an order-by, etc. They can also be type-safe in terms of referencing attributes as we will see in a bit. Users of the older Hibernate `org.hibernate.Criteria` query API will recognize the general approach, though we believe the JPA API to be superior as it represents a clean look at the lessons learned from that API.

Criteria queries are essentially an object graph, where each part of the graph represents an increasing (as we navigate down this graph) more atomic part of query. The first step in performing a criteria query is building this graph. The `javax.persistence.criteria.CriteriaBuilder` interface is the first thing with which you need to become acquainted to begin using criteria queries. Its role is that of a factory for all the individual pieces of the criteria. You obtain a `javax.persistence.criteria.CriteriaBuilder` instance by calling the `getCriteriaBuilder()` method of either `javax.persistence.EntityManagerFactory` or `javax.persistence.EntityManager`.

The next step is to obtain a `javax.persistence.criteria.CriteriaQuery`. This is accomplished using one of the three methods on `javax.persistence.criteria.CriteriaBuilder` for this purpose:

- `<T> CriteriaQuery<T> createQuery( Class<T> resultClass )`
- `CriteriaQuery<Tuple> createTupleQuery()`
- `CriteriaQuery<Object> createQuery()`

Each serves a different purpose depending on the expected type of the query results.

> Chapter 6 Criteria API of the JPA Specification already contains a decent amount of reference material pertaining to the various parts of a criteria query. So rather than duplicate all that content here, let's instead look at some of the more widely anticipated usages of the API.

16.1. Typed criteria queries

The type of the criteria query (aka the `<T>`) indicates the expected types in the query result. This might be an entity, an `Integer`, or any other object.

16.2. Selecting an entity

This is probably the most common form of query. The application wants to select entity instances.

Example 445. Selecting the root entity

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Person> persons = entityManager.createQuery( criteria ).getResultList();
```

JAVA

The example uses `createQuery()` passing in the `Person` class reference as the results of the query will be `Person` objects.

> The call to the `CriteriaQuery#select` method in this example is unnecessary because *root* will be the implied selection since we have only a single query root. It was done here only for completeness of an example.
>
> The `Person_.name` reference is an example of the static form of JPA Metamodel reference. We will use that form exclusively in this chapter. See the documentation for the Hibernate JPA Metamodel Generator (https://docs.jboss.org/hibernate/orm/5.2/topical/html/metamodelgen/MetamodelGenerator.html) for additional details on the JPA static Metamodel.

16.3. Selecting an expression

The simplest form of selecting an expression is selecting a particular attribute from an entity. But this expression might also represent an aggregation, a mathematical operation, etc.

Example 446. Selecting an attribute

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<String> criteria = builder.createQuery( String.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root.get( Person_.nickName ) );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<String> nickNames = entityManager.createQuery( criteria ).getResultList();
```
JAVA

In this example, the query is typed as `java.lang.String` because that is the anticipated type of the results (the type of the `Person#nickName` attribute is `java.lang.String`). Because a query might contain multiple references to the `Person` entity, attribute references always need to be qualified. This is accomplished by the `Root#get` method call.

16.4. Selecting multiple values

There are actually a few different ways to select multiple values using criteria queries. We will explore two options here, but an alternative recommended approach is to use tuples as described in Tuple criteria queries, or consider a wrapper query, see Selecting a wrapper for details.

Example 447. Selecting an array

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> root = criteria.from( Person.class );

Path<Long> idPath = root.get( Person_.id );
Path<String> nickNamePath = root.get( Person_.nickName);

criteria.select( builder.array( idPath, nickNamePath ) );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Object[]> idAndNickNames = entityManager.createQuery( criteria ).getResultList();
```
JAVA

Technically this is classified as a typed query, but you can see from handling the results that this is sort of misleading. Anyway, the expected result type here is an array.

The example then uses the array method of `javax.persistence.criteria.CriteriaBuilder` which explicitly combines individual selections into a `javax.persistence.criteria.CompoundSelection`.

Example 448. Selecting an array using `multiselect`

```java
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> root = criteria.from( Person.class );

Path<Long> idPath = root.get( Person_.id );
Path<String> nickNamePath = root.get( Person_.nickName);

criteria.multiselect( idPath, nickNamePath );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Object[]> idAndNickNames = entityManager.createQuery( criteria ).getResultList();
```

Just as we saw in Selecting an array we have a typed criteria query returning an `Object` array. Both queries are functionally equivalent. This second example uses the `multiselect()` method which behaves slightly differently based on the type given when the criteria query was first built, but, in this case, it says to select and return an *Object[]*.

16.5. Selecting a wrapper

Another alternative to Selecting multiple values is to instead select an object that will "wrap" the multiple values. Going back to the example query there, rather than returning an array of *[Person#id, Person#nickName]*, instead declare a class that holds these values and use that as a return object.

Example 449. Selecting a wrapper

```java
public class PersonWrapper {

    private final Long id;

    private final String nickName;

    public PersonWrapper(Long id, String nickName) {
        this.id = id;
        this.nickName = nickName;
    }

    public Long getId() {
        return id;
    }

    public String getNickName() {
        return nickName;
    }
}


CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<PersonWrapper> criteria = builder.createQuery( PersonWrapper.class );
Root<Person> root = criteria.from( Person.class );

Path<Long> idPath = root.get( Person_.id );
Path<String> nickNamePath = root.get( Person_.nickName);

criteria.select( builder.construct( PersonWrapper.class, idPath, nickNamePath ) );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<PersonWrapper> wrappers = entityManager.createQuery( criteria ).getResultList();
```

First, we see the simple definition of the wrapper object we will be using to wrap our result values. Specifically, notice the constructor and its argument types. Since we will be returning `PersonWrapper` objects, we use `PersonWrapper` as the type of our criteria query.

This example illustrates the use of the `javax.persistence.criteria.CriteriaBuilder` method construct which is used to build a wrapper expression. For every row in the result we are saying we would like a `PersonWrapper` instantiated with the remaining arguments by the matching constructor. This wrapper expression is then passed as the select.

16.6. Tuple criteria queries

A better approach to Selecting multiple values is to use either a wrapper (which we just saw in Selecting a wrapper) or using the `javax.persistence.Tuple` contract.

Example 450. Selecting a tuple

JAVA

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Tuple> criteria = builder.createQuery( Tuple.class );
Root<Person> root = criteria.from( Person.class );

Path<Long> idPath = root.get( Person_.id );
Path<String> nickNamePath = root.get( Person_.nickName);

criteria.multiselect( idPath, nickNamePath );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Tuple> tuples = entityManager.createQuery( criteria ).getResultList();

for ( Tuple tuple : tuples ) {
    Long id = tuple.get( idPath );
    String nickName = tuple.get( nickNamePath );
}

//or using indices
for ( Tuple tuple : tuples ) {
    Long id = (Long) tuple.get( 0 );
    String nickName = (String) tuple.get( 1 );
}
```

This example illustrates accessing the query results through the `javax.persistence.Tuple` interface. The example uses the explicit `createTupleQuery()` of `javax.persistence.criteria.CriteriaBuilder`. An alternate approach is to use `createQuery( Tuple.class )`.

Again we see the use of the `multiselect()` method, just like in Selecting an array using `multiselect`. The difference here is that the type of the `javax.persistence.criteria.CriteriaQuery` was defined as `javax.persistence.Tuple` so the compound selections, in this case, are interpreted to be the tuple elements.

The javax.persistence.Tuple contract provides three forms of access to the underlying elements:

typed

The Selecting a tuple example illustrates this form of access in the `tuple.get( idPath )` and `tuple.get( nickNamePath )` calls. This allows typed access to the underlying tuple values based on the `javax.persistence.TupleElement` expressions used to build the criteria.

positional

Allows access to the underlying tuple values based on the position. The simple *Object get(int position)* form is very similar to the access illustrated in Selecting an array and Selecting an array using `multiselect`. The *<X> X get(int position, Class<X> type* form allows typed positional access, but based on the explicitly supplied type which the tuple value must be type-assignable to.

aliased

Allows access to the underlying tuple values based an (optionally) assigned alias. The example query did not apply an alias. An alias would be applied via the alias method on `javax.persistence.criteria.Selection`. Just like `positional` access, there is both a typed (*Object get(String alias)*) and an untyped (*<X> X get(String alias, Class<X> type* form.

16.7. FROM clause

> A `CriteriaQuery` object defines a query over one or more entity, embeddable, or basic abstract schema types. The root objects of the query are entities, from which the other types are reached by navigation.
>
> — JPA Specification, section 6.5.2 Query Roots, pg 262

> All the individual parts of the FROM clause (roots, joins, paths) implement the `javax.persistence.criteria.From` interface.

16.8. Roots

Roots define the basis from which all joins, paths and attributes are available in the query. A root is always an entity type. Roots are defined and added to the criteria by the overloaded from methods on `javax.persistence.criteria.CriteriaQuery`:

Example 451. Root methods

```
<X> Root<X> from( Class<X> );

<X> Root<X> from( EntityType<X> );
```

JAVA

Example 452. Adding a root example

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );
```

JAVA

Criteria queries may define multiple roots, the effect of which is to create a Cartesian Product between the newly added root and the others. Here is an example defining a Cartesian Product between `Person` and `Partner` entities:

Example 453. Adding multiple roots example

```
JAVA
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Tuple> criteria = builder.createQuery( Tuple.class );

Root<Person> personRoot = criteria.from( Person.class );
Root<Partner> partnerRoot = criteria.from( Partner.class );
criteria.multiselect( personRoot, partnerRoot );

Predicate personRestriction = builder.and(
    builder.equal( personRoot.get( Person_.address ), address ),
    builder.isNotEmpty( personRoot.get( Person_.phones ) )
);
Predicate partnerRestriction = builder.and(
    builder.like( partnerRoot.get( Partner_.name ), prefix ),
    builder.equal( partnerRoot.get( Partner_.version ), 0 )
);
criteria.where( builder.and( personRestriction, partnerRestriction ) );

List<Tuple> tuples = entityManager.createQuery( criteria ).getResultList();
```

16.9. Joins

Joins allow navigation from other `javax.persistence.criteria.From` to either association or embedded attributes. Joins are created by the numerous overloaded join methods of the `javax.persistence.criteria.From` interface.

Example 454. Join example

```
JAVA
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Phone> criteria = builder.createQuery( Phone.class );
Root<Phone> root = criteria.from( Phone.class );

// Phone.person is a @ManyToOne
Join<Phone, Person> personJoin = root.join( Phone_.person );
// Person.addresses is an @ElementCollection
Join<Person, String> addressesJoin = personJoin.join( Person_.addresses );

criteria.where( builder.isNotEmpty( root.get( Phone_.calls ) ) );

List<Phone> phones = entityManager.createQuery( criteria ).getResultList();
```

16.10. Fetches

Just like in HQL and JPQL, criteria queries can specify that associated data be fetched along with the owner. Fetches are created by the numerous overloaded fetch methods of the `javax.persistence.criteria.From` interface.

Example 455. Join fetch example

JAVA

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Phone> criteria = builder.createQuery( Phone.class );
Root<Phone> root = criteria.from( Phone.class );

// Phone.person is a @ManyToOne
Fetch<Phone, Person> personFetch = root.fetch( Phone_.person );
// Person.addresses is an @ElementCollection
Fetch<Person, String> addressesJoin = personFetch.fetch( Person_.addresses );

criteria.where( builder.isNotEmpty( root.get( Phone_.calls ) ) );

List<Phone> phones = entityManager.createQuery( criteria ).getResultList();
```

> Technically speaking, embedded attributes are always fetched with their owner. However in order to define the fetching of *Phone#addresses* we needed a `javax.persistence.criteria.Fetch` because element collections are `LAZY` by default.

16.11. Path expressions

> Roots, joins and fetches are themselves paths as well.

16.12. Using parameters

Example 456. Parameters example

```java
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );

ParameterExpression<String> nickNameParameter = builder.parameter( String.class );
criteria.where( builder.equal( root.get( Person_.nickName ), nickNameParameter ) );

TypedQuery<Person> query = entityManager.createQuery( criteria );
query.setParameter( nickNameParameter, "JD" );
List<Person> persons = query.getResultList();
```

Use the parameter method of `javax.persistence.criteria.CriteriaBuilder` to obtain a parameter reference. Then use the parameter reference to bind the parameter value to the `javax.persistence.Query`.

16.13. Using group by

Example 457. Group by example

```java
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Tuple> criteria = builder.createQuery( Tuple.class );
Root<Person> root = criteria.from( Person.class );

criteria.groupBy(root.get("address"));
criteria.multiselect(root.get("address"), builder.count(root));

List<Tuple> tuples = entityManager.createQuery( criteria ).getResultList();

for ( Tuple tuple : tuples ) {
    String name = (String) tuple.get( 0 );
    Long count = (Long) tuple.get( 1 );
}
```

17. Native SQL Queries

You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features such as window functions, Common Table Expressions (CTE) or the `CONNECT BY` option in Oracle. It also provides a clean migration path from a direct SQL/JDBC based application to Hibernate/JPA. Hibernate also allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and retrieve operations.

17.1. Creating a native query using JPA

Execution of native SQL queries is controlled via the `SQLQuery` interface, which is obtained by calling `Session.createSQLQuery()`. The following sections describe how to use this API for querying.

17.2. Scalar queries

The most basic SQL query is to get a list of scalars (column) values.

Example 458. JPA native query selecting all columns

```
List<Object[]> persons = entityManager.createNativeQuery(
    "SELECT * FROM Person" )
.getResultList();
```

JAVA

Example 459. JPA native query with a custom column selection

```
List<Object[]> persons = entityManager.createNativeQuery(
    "SELECT id, name FROM Person" )
.getResultList();

for(Object[] person : persons) {
    Number id = (Number) person[0];
    String name = (String) person[1];
}
```

JAVA

Example 460. Hibernate native query selecting all columns

```
List<Object[]> persons = session.createSQLQuery(
    "SELECT * FROM Person" )
.list();
```

JAVA

Example 461. Hibernate native query with a custom column selection

```java
List<Object[]> persons = session.createSQLQuery(
    "SELECT id, name FROM Person" )
.list();

for(Object[] person : persons) {
    Number id = (Number) person[0];
    String name = (String) person[1];
}
```

These will return a `List` of `Object` arrays (`Object[]`) with scalar values for each column in the `PERSON` table. Hibernate will use `java.sql.ResultSetMetadata` to deduce the actual order and types of the returned scalar values.

To avoid the overhead of using `ResultSetMetadata` , or simply to be more explicit in what is returned, one can use `addScalar()` :

Example 462. Hibernate native query with explicit result set selection

```java
List<Object[]> persons = session.createSQLQuery(
    "SELECT * FROM Person" )
.addScalar( "id", LongType.INSTANCE )
.addScalar( "name", StringType.INSTANCE )
.list();

for(Object[] person : persons) {
    Long id = (Long) person[0];
    String name = (String) person[1];
}
```

Although it still returns an `Object` arrays, this query will not use the `ResultSetMetadata` anymore since it explicitly gets the `id` and `name` columns as respectively a `BigInteger` and a `String` from the underlying `ResultSet` . This also means that only these two columns will be returned, even though the query is still using `*` and the `ResultSet` contains more than the three listed columns.

It is possible to leave out the type information for all or some of the scalars.

Example 463. Hibernate native query with result set selection that's a partially explicit

```java
List<Object[]> persons = session.createSQLQuery(
    "SELECT * FROM Person" )
.addScalar( "id", LongType.INSTANCE )
.addScalar( "name" )
.list();

for(Object[] person : persons) {
    Long id = (Long) person[0];
    String name = (String) person[1];
}
```

This is essentially the same query as before, but now `ResultSetMetaData` is used to determine the type of `name`, where as the type of `id` is explicitly specified.

How the `java.sql.Types` returned from `ResultSetMetaData` is mapped to Hibernate types is controlled by the `Dialect`. If a specific type is not mapped, or does not result in the expected type, it is possible to customize it via calls to `registerHibernateType` in the Dialect.

17.3. Entity queries

The above queries were all about returning scalar values, basically returning the *raw* values from the `ResultSet`.

Example 464. JPA native query selecting entities

```java
List<Person> persons = entityManager.createNativeQuery(
    "SELECT * FROM Person", Person.class )
.getResultList();
```

Example 465. Hibernate native query selecting entities

```java
List<Person> persons = session.createSQLQuery(
    "SELECT * FROM Person" )
.addEntity( Person.class )
.list();
```

Assuming that `Person` is mapped as a class with the columns `id`, `name`, `nickName`, `address`, `createdOn` and `version`, the following query will also return a `List` where each element is a `Person` entity.

Example 466. JPA native query selecting entities with explicit result set

```
List<Person> persons = entityManager.createNativeQuery(
    "SELECT id, name, nickName, address, createdOn, version " +
    "FROM Person", Person.class )
.getResultList();
```
JAVA

Example 467. Hibernate native query selecting entities with explicit result set

```
List<Person> persons = session.createSQLQuery(
    "SELECT id, name, nickName, address, createdOn, version " +
    "FROM Person" )
.addEntity( Person.class )
.list();
```
JAVA

17.4. Handling associations and collections

If the entity is mapped with a `many-to-one` or a child-side `one-to-one` to another entity, it is required to also return this when performing the native query, otherwise a database specific *column not found* error will occur.

Example 468. JPA native query selecting entities with many-to-one association

```
List<Phone> phones = entityManager.createNativeQuery(
    "SELECT id, phone_number, phone_type, person_id " +
    "FROM Phone", Phone.class )
.getResultList();
```
JAVA

Example 469. Hibernate native query selecting entities with many-to-one association

```
List<Phone> phones = session.createSQLQuery(
    "SELECT id, phone_number, phone_type, person_id " +
    "FROM Phone" )
.addEntity( Phone.class )
.list();
```
JAVA

This will allow the `Phone#person` to function properly.

> The additional columns will automatically be returned when using the `*` notation.

It is possible to eagerly join the `Phone` and the `Person` entities to avoid the possible extra roundtrip for initializing the `many-to-one` association.

Example 470. JPA native query selecting entities with joined many-to-one association

JAVA

```
List<Phone> phones = entityManager.createNativeQuery(
    "SELECT * " +
    "FROM Phone ph " +
    "JOIN Person pr ON ph.person_id = pr.id", Phone.class )
.getResultList();

for(Phone phone : phones) {
    Person person = phone.getPerson();
}
```

SQL

```
SELECT id ,
       number ,
       type ,
       person_id
FROM   phone
```

Example 471. Hibernate native query selecting entities with joined many-to-one association

JAVA

```
List<Object[]> tuples = session.createSQLQuery(
    "SELECT * " +
    "FROM Phone ph " +
    "JOIN Person pr ON ph.person_id = pr.id" )
.addEntity("phone", Phone.class )
.addJoin( "pr", "phone.person")
.list();

for(Object[] tuple : tuples) {
    Phone phone = (Phone) tuple[0];
    Person person = (Person) tuple[1];
}
```

```
SELECT id ,
       number ,
       type ,
       person_id
FROM   phone
```

SQL

> As seen in the associated SQL query, Hibernate manages to construct the entity hierarchy without requiring any extra database roundtrip.

By default, when using the `addJoin()` method, the result set will contain both entities that are joined. To construct the entity hierarchy, you need to use a `ROOT_ENTITY` or `DISTINCT_ROOT_ENTITY` `ResultTransformer`.

Example 472. Hibernate native query selecting entities with joined many-to-one association and `ResultTransformer`

```
List<Person> persons = session.createSQLQuery(
    "SELECT * " +
    "FROM Phone ph " +
    "JOIN Person pr ON ph.person_id = pr.id" )
.addEntity("phone", Phone.class )
.addJoin( "pr", "phone.person")
.setResultTransformer( Criteria.ROOT_ENTITY )
.list();

for(Person person : persons) {
    person.getPhones();
}
```

JAVA

> Because of the `ROOT_ENTITY ResultTransformer`, this query will return the parent-side as root entities.

Notice that you added an alias name *pr* to be able to specify the target property path of the join. It is possible to do the same eager joining for collections (e.g. the `Phone#calls one-to-many` association).

Example 473. JPA native query selecting entities with joined one-to-many association

JAVA

```
List<Phone> phones = entityManager.createNativeQuery(
    "SELECT * " +
    "FROM Phone ph " +
    "JOIN phone_call c ON c.phone_id = ph.id", Phone.class )
.getResultList();

for(Phone phone : phones) {
    List<Call> calls = phone.getCalls();
}
```

SQL

```
SELECT *
FROM phone ph
JOIN call c ON c.phone_id = ph.id
```

Example 474. Hibernate native query selecting entities with joined one-to-many association

JAVA

```
List<Object[]> tuples = session.createSQLQuery(
    "SELECT * " +
    "FROM Phone ph " +
    "JOIN phone_call c ON c.phone_id = ph.id" )
.addEntity("phone", Phone.class )
.addJoin( "c", "phone.calls")
.list();

for(Object[] tuple : tuples) {
    Phone phone = (Phone) tuple[0];
    Call call = (Call) tuple[1];
}
```

```sql
SELECT *
FROM phone ph
JOIN call c ON c.phone_id = ph.id
```

At this stage you are reaching the limits of what is possible with native queries, without starting to enhance the sql queries to make them usable in Hibernate. Problems can arise when returning multiple entities of the same type or when the default alias/column names are not enough.

17.5. Returning multiple entities

Until now, the result set column names are assumed to be the same as the column names specified in the mapping document. This can be problematic for SQL queries that join multiple tables since the same column names can appear in more than one table.

Column alias injection is needed in the following query which otherwise throws `NonUniqueDiscoveredSqlAliasException`.

Example 475. JPA native query selecting entities with the same column names

```java
List<Object> entities = entityManager.createNativeQuery(
    "SELECT * " +
    "FROM Person pr, Partner pt " +
    "WHERE pr.name = pt.name" )
.getResultList();
```

Example 476. Hibernate native query selecting entities with the same column names

```java
List<Object> entities = session.createSQLQuery(
        "SELECT * " +
                "FROM Person pr, Partner pt " +
                "WHERE pr.name = pt.name" )
        .list();
```

The query was intended to return all `Person` and `Partner` instances with the same name. The query fails because there is a conflict of names since the two entities are mapped to the same column names (e.g. `id`, `name`, `version`). Also, on some databases the returned column aliases will most likely be on the form `pr.id`, `pr.name`, etc. which are not equal to the columns specified in the mappings (`id` and `name`).

The following form is not vulnerable to column name duplication:

Example 477. Hibernate native query selecting entities with the same column names and aliases

```java
List<Object> entities = session.createSQLQuery(
    "SELECT {pr.*}, {pt.*} " +
    "FROM Person pr, Partner pt " +
    "WHERE pr.name = pt.name" )
.addEntity( "pr", Person.class)
.addEntity( "pt", Partner.class)
.list();
```

> There's no such equivalent in JPA because the `Query` interface doesn't define an `addEntity` method equivalent.

The `{pr.}` **and** `{pt. }` notation used above is shorthand for "all properties". Alternatively, you can list the columns explicitly, but even in this case Hibernate injects the SQL column aliases for each property. The placeholder for a column alias is just the property name qualified by the table alias.

17.6. Alias and property references

In most cases the above alias injection is needed. For queries relating to more complex mappings, like composite properties, inheritance discriminators, collections etc., you can use specific aliases that allow Hibernate to inject the proper aliases.

The following table shows the different ways you can use the alias injection. Please note that the alias names in the result are simply examples, each alias will have a unique and probably different name when used.

Table 8. Alias injection names

Description	Syntax	Example
A simple property	`{[aliasname].[propertyname]`	`A_NAME as {item.name}`
A composite property	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
Discriminator of an entity	`{[aliasname].class}`	`DISC as {item.class}`

Description	Syntax	Example
All properties of an entity	`{[aliasname].*}`	`{item.*}`
A collection key	`{[aliasname].key}`	`ORGID as {coll.key}`
The id of an collection	`{[aliasname].id}`	`EMPID as {coll.id}`
The element of an collection	`{[aliasname].element}`	`XID as {coll.element}`
property of the element in the collection	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`
All properties of the element in the collection	`{[aliasname].element.*}`	`{coll.element.*}`
All properties of the collection	`{[aliasname].*}`	`{coll.*}`

17.7. Returning DTOs (Data Transfer Objects)

It is possible to apply a `ResultTransformer` to native SQL queries, allowing it to return non-managed entities.

Example 478. Hibernate native query selecting DTOs

```java
public class PersonSummaryDTO {

    private Number id;

    private String name;

    public Number getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

List<PersonSummaryDTO> dtos = session.createSQLQuery(
    "SELECT p.id as \"id\", p.name as \"name\" " +
    "FROM Person p")
.setResultTransformer( Transformers.aliasToBean( PersonSummaryDTO.class ) )
.list();
```

> There's no such equivalent in JPA because the `Query` interface doesn't define a `setResultTransformer` method equivalent.

The above query will return a list of `PersonSummaryDTO` which has been instantiated and injected the values of `id` and `name` into its corresponding properties or fields.

17.8. Handling inheritance

Native SQL queries which query for entities that are mapped as part of an inheritance must include all properties for the base class and all its subclasses.

Example 479. Hibernate native query selecting subclasses

```
List<CreditCardPayment> payments = session.createSQLQuery(
    "SELECT * " +
    "FROM Payment p " +
    "JOIN CreditCardPayment cp on cp.id = p.id" )
.addEntity( CreditCardPayment.class )
.list();
```

JAVA

> There's no such equivalent in JPA because the `Query` interface doesn't define an `addEntity` method equivalent.

17.9. Parameters

Native SQL queries support positional as well as named parameters:

Example 480. JPA native query with parameters

```
List<Person> persons = entityManager.createNativeQuery(
    "SELECT * " +
    "FROM Person " +
    "WHERE name like :name", Person.class )
.setParameter("name", "J%")
.getResultList();
```

JAVA

Example 481. Hibernate native query with parameters

```
List<Person> persons = session.createSQLQuery(
    "SELECT * " +
    "FROM Person " +
    "WHERE name like :name" )
.addEntity( Person.class )
.setParameter("name", "J%")
.list();
```

JAVA

17.10. Named SQL queries

Named SQL queries can also be defined during mapping and called in exactly the same way as a named HQL query. In this case, you do *not* need to call `addEntity()` anymore.

JPA defines the `javax.persistence.NamedNativeQuery` annotation for this purpose, and the Hibernate `org.hibernate.annotations.NamedNativeQuery` annotation extends it and adds the following attributes:

`flushMode()`
The flush mode for the query. By default, it uses the current Persistence Context flush mode.

`cacheable()`
Whether the query (results) is cacheable or not. By default, queries are not cached.

`cacheRegion()`
If the query results are cacheable, name the query cache region to use.

`fetchSize()`
The number of rows fetched by the JDBC Driver per database trip. The default value is given by the JDBC driver.

`timeout()`
The query timeout (in seconds). By default, there's no timeout.

`callable()`
Does the SQL query represent a call to a procedure/function? Default is false.

`comment()`
A comment added to the SQL query for tuning the execution plan.

`cacheMode()`
The cache mode used for this query. This refers to entities/collections returned by the query. The default value is `CacheModeType.NORMAL`.

`readOnly()`
Whether the results should be read-only. By default, queries are not read-only so entities are stored in the Persistence Context.

17.10.1. Named SQL queries selecting scalar values

To fetch a single column of given table, the named query looks as follows:

Example 482. Single scalar value `NamedNativeQuery`

```
@NamedNativeQuery(
    name = "find_person_name",
    query =
        "SELECT name " +
        "FROM Person "
),
```

JAVA

Example 483. JPA named native query selecting a scalar value

```
List<String> names = entityManager.createNamedQuery(
    "find_person_name" )
.getResultList();
```

JAVA

Example 484. Hibernate named native query selecting a scalar value

```
List<String> names = session.getNamedQuery(
    "find_person_name" )
.list();
```

JAVA

Selecting multiple scalar values is done like this:

Example 485. Multiple scalar values `NamedNativeQuery`

```
@NamedNativeQuery(
    name = "find_person_name_and_nickName",
    query =
        "SELECT " +
        "   name, " +
        "   nickName " +
        "FROM Person "
),
```

JAVA

Without specifying an explicit result type, Hibernate will assume an `Object` array:

Example 486. JPA named native query selecting multiple scalar values

```java
List<Object[]> tuples = entityManager.createNamedQuery(
    "find_person_name_and_nickName" )
.getResultList();

for(Object[] tuple : tuples) {
    String name = (String) tuple[0];
    String nickName = (String) tuple[1];
}
```

Example 487. Hibernate named native query selecting multiple scalar values

```java
List<Object[]> tuples = session.getNamedQuery(
    "find_person_name_and_nickName" )
.list();

for(Object[] tuple : tuples) {
    String name = (String) tuple[0];
    String nickName = (String) tuple[1];
}
```

It's possible to use a DTO to store the resulting scalar values:

Example 488. DTO to store multiple scalar values

```java
public class PersonNames {

    private final String name;

    private final String nickName;

    public PersonNames(String name, String nickName) {
        this.name = name;
        this.nickName = nickName;
    }

    public String getName() {
        return name;
    }

    public String getNickName() {
        return nickName;
    }
}
```

Example 489. Multiple scalar values `NamedNativeQuery` *with* `ConstructorResult`

```
@NamedNativeQuery(
    name = "find_person_name_and_nickName_dto",
    query =
        "SELECT " +
        "   name, " +
        "   nickName " +
        "FROM Person ",
    resultSetMapping = "name_and_nickName_dto"
),
@SqlResultSetMapping(
    name = "name_and_nickName_dto",
    classes = @ConstructorResult(
        targetClass = PersonNames.class,
        columns = {
            @ColumnResult(name = "name"),
            @ColumnResult(name = "nickName")
        }
    )
)
```

JAVA

Example 490. JPA named native query selecting multiple scalar values into a DTO

```
List<PersonNames> personNames = entityManager.createNamedQuery(
    "find_person_name_and_nickName_dto" )
.getResultList();
```

JAVA

Example 491. Hibernate named native query selecting multiple scalar values into a DTO

```
List<PersonNames> personNames = session.getNamedQuery(
    "find_person_name_and_nickName_dto" )
.list();
```

JAVA

17.10.2. Named SQL queries selecting entities

Considering the following named query:

Example 492. Single-entity `NamedNativeQuery`

```
@NamedNativeQuery(
    name = "find_person_by_name",
    query =
        "SELECT " +
        "   p.id AS \"id\", " +
        "   p.name AS \"name\", " +
        "   p.nickName AS \"nickName\", " +
        "   p.address AS \"address\", " +
        "   p.createdOn AS \"createdOn\", " +
        "   p.version AS \"version\" " +
        "FROM Person p " +
        "WHERE p.name LIKE :name",
    resultClass = Person.class
),
```

JAVA

The result set 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.

Executing this named native query can be done as follows:

Example 493. JPA named native entity query

```
List<Person> persons = entityManager.createNamedQuery(
    "find_person_by_name" )
.setParameter("name", "J%")
.getResultList();
```

JAVA

Example 494. Hibernate named native entity query

```
List<Person> persons = session.getNamedQuery(
    "find_person_by_name" )
.setParameter("name", "J%")
.list();
```

JAVA

To join multiple entities, you need to use a `SqlResultSetMapping` for each entity the SQL query is going to fetch.

Example 495. Joined-entities `NamedNativeQuery`

JAVA

```
@NamedNativeQuery(
    name = "find_person_with_phones_by_name",
    query =
        "SELECT " +
        "   pr.id AS \"pr.id\", " +
        "   pr.name AS \"pr.name\", " +
        "   pr.nickName AS \"pr.nickName\", " +
        "   pr.address AS \"pr.address\", " +
        "   pr.createdOn AS \"pr.createdOn\", " +
        "   pr.version AS \"pr.version\", " +
        "   ph.id AS \"ph.id\", " +
        "   ph.person_id AS \"ph.person_id\", " +
        "   ph.phone_number AS \"ph.number\", " +
        "   ph.phone_type AS \"ph.type\" " +
        "FROM Person pr " +
        "JOIN Phone ph ON pr.id = ph.person_id " +
        "WHERE pr.name LIKE :name",
    resultSetMapping = "person_with_phones"
)
 @SqlResultSetMapping(
     name = "person_with_phones",
     entities = {
         @EntityResult(
             entityClass = Person.class,
             fields = {
                 @FieldResult( name = "id", column = "pr.id" ),
                 @FieldResult( name = "name", column = "pr.name" ),
                 @FieldResult( name = "nickName", column = "pr.nickName" ),
                 @FieldResult( name = "address", column = "pr.address" ),
                 @FieldResult( name = "createdOn", column = "pr.createdOn" ),
                 @FieldResult( name = "version", column = "pr.version" ),
             }
         ),
         @EntityResult(
             entityClass = Phone.class,
             fields = {
                 @FieldResult( name = "id", column = "ph.id" ),
                 @FieldResult( name = "person", column = "ph.person_id" ),
                 @FieldResult( name = "number", column = "ph.number" ),
                 @FieldResult( name = "type", column = "ph.type" ),
             }
         )
     }
 ),
```

Example 496. JPA named native entity query with joined associations

```java
List<Object[]> tuples = entityManager.createNamedQuery(
    "find_person_with_phones_by_name" )
.setParameter("name", "J%")
.getResultList();

for(Object[] tuple : tuples) {
    Person person = (Person) tuple[0];
    Phone phone = (Phone) tuple[1];
}
```

Example 497. Hibernate named native entity query with joined associations

```java
List<Object[]> tuples = session.getNamedQuery(
    "find_person_with_phones_by_name" )
.setParameter("name", "J%")
.list();

for(Object[] tuple : tuples) {
    Person person = (Person) tuple[0];
    Phone phone = (Phone) tuple[1];
}
```

Finally, 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. For this example, the following entities are going to be used:

Example 498. Entity associations with composite keys and named native queries

JAVA

```
@Embeddable
public class Dimensions {

    private int length;

    private int width;

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }
}

@Embeddable
public class Identity implements Serializable {

    private String firstname;

    private String lastname;

    public String getFirstname() {
        return firstname;
    }

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

    public String getLastname() {
        return lastname;
    }

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

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

        final Identity identity = (Identity) o;

        if ( !firstname.equals( identity.firstname ) ) return false;
        if ( !lastname.equals( identity.lastname ) ) return false;
```

```
        return true;
    }

    public int hashCode() {
        int result;
        result = firstname.hashCode();
        result = 29 * result + lastname.hashCode();
        return result;
    }
}

@Entity
public class Captain {

    @EmbeddedId
    private Identity id;

    public Identity getId() {
        return id;
    }

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

@Entity
@NamedNativeQueries({
    @NamedNativeQuery(name = "find_all_spaceships",
        query =
            "SELECT " +
            "   name as \"name\", " +
            "   model, " +
            "   speed, " +
            "   lname as lastn, " +
            "   fname as firstn, " +
            "   length, " +
            "   width, " +
            "   length * width as surface, " +
            "   length * width * 10 as volume " +
            "FROM SpaceShip",
        resultSetMapping = "spaceship"
    )
})
@SqlResultSetMapping(
    name = "spaceship",
    entities = @EntityResult(
        entityClass = SpaceShip.class,
        fields = {
            @FieldResult(name = "name", column = "name"),
            @FieldResult(name = "model", column = "model"),
            @FieldResult(name = "speed", column = "speed"),
            @FieldResult(name = "captain.lastname", column = "lastn"),
            @FieldResult(name = "captain.firstname", column = "firstn"),
            @FieldResult(name = "dimensions.length", column = "length"),
            @FieldResult(name = "dimensions.width", column = "width"),
        }
    ),
```

```
    columns = {
        @ColumnResult(name = "surface"),
        @ColumnResult(name = "volume")
    }
)
public class SpaceShip {

    @Id
    private String name;

    private String model;

    private double speed;

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

    private Dimensions dimensions;

    public String getName() {
        return name;
    }

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

    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 Captain getCaptain() {
        return captain;
    }

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

    public Dimensions getDimensions() {
        return dimensions;
    }
```

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

Example 499. JPA named native entity query with joined associations and composite keys

```
List<Object[]> tuples = entityManager.createNamedQuery(
    "find_all_spaceships" )
.getResultList();

for(Object[] tuple : tuples) {
    SpaceShip spaceShip = (SpaceShip) tuple[0];
    Number surface = (Number) tuple[1];
    Number volume = (Number) tuple[2];
}
```

JAVA

Example 500. Hibernate named native entity query with joined associations and composite keys

```
List<Object[]> tuples = session.getNamedQuery(
    "find_all_spaceships" )
.list();

for(Object[] tuple : tuples) {
    SpaceShip spaceShip = (SpaceShip) tuple[0];
    Number surface = (Number) tuple[1];
    Number volume = (Number) tuple[2];
}
```

JAVA

17.11. Resolving global catalog and schema in native SQL queries

When using multiple database catalogs and schemas, Hibernate offers the possibility of setting a global catalog or schema so that you don't have to declare it explicitly for every entity.

Example 501. Setting global catalog and schema

```
<property name="hibernate.default_catalog" value="crm"/>
<property name="hibernate.default_schema" value="analytics"/>
```

XML

This way, we can imply the global **crm** catalog and **analytics** schema in every JPQL, HQL or Criteria API query.

However, for native queries, the SQL query is passed as is, therefore you need to explicitly set the global catalog and schema whenever you are referencing a database table. Fortunately, Hibernate allows you to resolve the current global catalog and schema using the following placeholders:

`{h-catalog}`
resolves the current `hibernate.default_catalog` configuration property value.

`{h-schema}`
resolves the current `hibernate.default_schema` configuration property value.

`{h-domain}`
resolves the current `hibernate.default_catalog` and `hibernate.default_schema` configuration property values (e.g. catalog.schema).

Withe these placeholders, you can imply the catalog, schema, or both catalog and schema for every native query.

So, when running the following native query:

```
JAVA
@NamedNativeQuery(
    name = "last_30_days_hires",
    query =
        "select * " +
        "from {h-domain}person " +
        "where age(hired_on) < '30 days'",
    resultClass = Person.class
)
```

Hibernate is going to resolve the `{h-domain}` placeholder according to the values of the default catalog and schema:

```
SQL
SELECT *
FROM   crm.analytics.person
WHERE  age(hired_on) < '30 days'
```

17.12. Using stored procedures for querying

Hibernate provides support for queries via stored procedures and functions. A stored procedure arguments are declared using the `IN` parameter type, and the result can be either marked with an `OUT` parameter type, a `REF_CURSOR` or it could just return the result like a function.

Example 502. MySQL stored procedure with `OUT` parameter type

```
statement.executeUpdate(
    "CREATE PROCEDURE sp_count_phones (" +
    "   IN personId INT, " +
    "   OUT phoneCount INT " +
    ") " +
    "BEGIN " +
    "    SELECT COUNT(*) INTO phoneCount " +
    "    FROM Phone p " +
    "    WHERE p.person_id = personId; " +
    "END"
);
```

JAVA

To use this stored procedure, you can execute the following JPA 2.1 query:

Example 503. Calling a MySQL stored procedure with `OUT` parameter type using JPA

```
StoredProcedureQuery query = entityManager.createStoredProcedureQuery( "sp_count_phones");
query.registerStoredProcedureParameter( "personId", Long.class, ParameterMode.IN);
query.registerStoredProcedureParameter( "phoneCount", Long.class, ParameterMode.OUT);

query.setParameter("personId", 1L);

query.execute();
Long phoneCount = (Long) query.getOutputParameterValue("phoneCount");
```

JAVA

Example 504. Calling a MySQL stored procedure with `OUT` parameter type using Hibernate

```
Session session = entityManager.unwrap( Session.class );

ProcedureCall call = session.createStoredProcedureCall( "sp_count_phones" );
call.registerParameter( "personId", Long.class, ParameterMode.IN ).bindValue( 1L );
call.registerParameter( "phoneCount", Long.class, ParameterMode.OUT );

Long phoneCount = (Long) call.getOutputs().getOutputParameterValue( "phoneCount" );
assertEquals( Long.valueOf( 2 ), phoneCount );
```

JAVA

If the stored procedure outputs the result directly without an `OUT` parameter type:

Example 505. MySQL stored procedure without an `OUT` parameter type

```
statement.executeUpdate(
    "CREATE PROCEDURE sp_phones(IN personId INT) " +
    "BEGIN " +
    "    SELECT *  " +
    "    FROM Phone   " +
    "    WHERE person_id = personId;  " +
    "END"
);
```
JAVA

You can retrieve the results of the aforementioned MySQL stored procedure as follows:

Example 506. Calling a MySQL stored procedure and fetching the result set without an `OUT` parameter type using JPA

```
StoredProcedureQuery query = entityManager.createStoredProcedureQuery( "sp_phones");
query.registerStoredProcedureParameter( 1, Long.class, ParameterMode.IN);

query.setParameter(1, 1L);

List<Object[]> personComments = query.getResultList();
```
JAVA

Example 507. Calling a MySQL stored procedure and fetching the result set without an `OUT` parameter type using Hibernate

```
Session session = entityManager.unwrap( Session.class );

ProcedureCall call = session.createStoredProcedureCall( "sp_phones" );
call.registerParameter( 1, Long.class, ParameterMode.IN ).bindValue( 1L );

Output output = call.getOutputs().getCurrent();

List<Object[]> personComments = ( (ResultSetOutput) output ).getResultList();
```
JAVA

For a `REF_CURSOR` result sets, we'll consider the following Oracle stored procedure:

Example 508. Oracle `REF_CURSOR` stored procedure

```
statement.executeUpdate(
    "CREATE OR REPLACE PROCEDURE sp_person_phones ( " +
    "   personId IN NUMBER, " +
    "   personPhones OUT SYS_REFCURSOR ) " +
    "AS  " +
    "BEGIN " +
    "    OPEN personPhones FOR " +
    "    SELECT *" +
    "    FROM phone " +
    "    WHERE person_id = personId; " +
    "END;"
);
```
JAVA

> `REF_CURSOR` result sets are only supported by Oracle and PostgreSQL because other database systems JDBC drivers don't support this feature.

This function can be called using the standard Java Persistence API:

Example 509. Calling an Oracle `REF_CURSOR` stored procedure using JPA

```
StoredProcedureQuery query = entityManager.createStoredProcedureQuery( "sp_person_phones" );
query.registerStoredProcedureParameter( 1, Long.class, ParameterMode.IN );
query.registerStoredProcedureParameter( 2, Class.class, ParameterMode.REF_CURSOR );
query.setParameter( 1, 1L );

query.execute();
List<Object[]> postComments = query.getResultList();
```
JAVA

Example 510. Calling an Oracle `REF_CURSOR` stored procedure using Hibernate

```
Session session = entityManager.unwrap(Session.class);

ProcedureCall call = session.createStoredProcedureCall( "sp_person_phones");
call.registerParameter(1, Long.class, ParameterMode.IN).bindValue(1L);
call.registerParameter(2, Class.class, ParameterMode.REF_CURSOR);

Output output = call.getOutputs().getCurrent();
List<Object[]> postComments = ( (ResultSetOutput) output ).getResultList();
assertEquals(2, postComments.size());
```

JAVA

If the database defines an SQL function:

Example 511. MySQL function

```
statement.executeUpdate(
    "CREATE FUNCTION fn_count_phones(personId integer)  " +
    "RETURNS integer " +
    "DETERMINISTIC " +
    "READS SQL DATA " +
    "BEGIN " +
    "    DECLARE phoneCount integer; " +
    "    SELECT COUNT(*) INTO phoneCount " +
    "    FROM Phone p " +
    "    WHERE p.person_id = personId; " +
    "    RETURN phoneCount; " +
    "END"
);
```

JAVA

Because the current `StoredProcedureQuery` implementation doesn't yet support SQL functions, we need to use the JDBC syntax.

> This limitation is acknowledged and will be addressed by the HHH-10530 (https://hibernate.atlassian.net/browse/HHH-10530) issue.

Example 512. Calling a MySQL function

```java
final AtomicReference<Integer> phoneCount = new AtomicReference<>();
Session session = entityManager.unwrap( Session.class );
session.doWork( connection -> {
    try (CallableStatement function = connection.prepareCall(
            "{ ? = call fn_count_phones(?) }" )) {
        function.registerOutParameter( 1, Types.INTEGER );
        function.setInt( 2, 1 );
        function.execute();
        phoneCount.set( function.getInt( 1 ) );
    }
} );
```

> Stored procedure queries cannot be paged with `setFirstResult()/setMaxResults()`.
>
> Since these servers can return multiple result sets and update counts, Hibernate will iterate the results and take the first result that is a result set as its return value, so everything else will be discarded.

For SQL Server, if you can enable `SET NOCOUNT ON` in your procedure it will probably be more efficient, but this is not a requirement.

17.13. Custom SQL for create, update, and delete

Hibernate can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or individual column level. This section describes statement overrides. For columns, see Column transformers: read and write expressions.

The following example shows how to define custom SQL operations using annotations. `@SQLInsert`, `@SQLUpdate` and `@SQLDelete` override the INSERT, UPDATE, DELETE statements of a given entity. For the SELECT clause, a `@Loader` must be defined along with a `@NamedNativeQuery` used for loading the underlying table record.

For collections, Hibernate allows defining a custom `@SQLDeleteAll` which is used for removing all child records associated with a given parent entity. To filter collections, the `@Where` annotation allows customizing the underlying SQL WHERE clause.

Example 513. Custom CRUD

JAVA

```
@Entity(name = "Person")
@SQLInsert(
    sql = "INSERT INTO person (name, id, valid) VALUES (?, ?, true) ",
    check = ResultCheckStyle.COUNT
)
@SQLUpdate(
    sql = "UPDATE person SET name = ? where id = ? ")
@SQLDelete(
    sql = "UPDATE person SET valid = false WHERE id = ? ")
@Loader(namedQuery = "find_valid_person")
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "find_valid_person",
        query = "SELECT id, name " +
                "FROM person " +
                "WHERE id = ? and valid = true",
        resultClass = Person.class
    )
})
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ElementCollection
    @SQLInsert(
        sql = "INSERT INTO person_phones (person_id, phones, valid) VALUES (?, ?, true) ")
    @SQLDeleteAll(
        sql = "UPDATE person_phones SET valid = false WHERE person_id = ?")
    @Where( clause = "valid = true" )
    private List<String> phones = new ArrayList<>();

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public List<String> getPhones() {
        return phones;
    }
}
```

In the example above, the entity is mapped so that entries are soft-deleted (the records are not removed from the database, but instead, a flag marks the row validity). The `Person` entity benefits from custom INSERT, UPDATE, and DELETE statements which update the `valid` column accordingly. The custom `@Loader` is used to retrieve only `Person` rows that are valid.

The same is done for the `phones` collection. The `@SQLDeleteAll` and the `SQLInsert` queries are used whenever the collection is modified.

> You also call a store procedure using the custom CRUD statements; the only requirement is to set the `callable` attribute to `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 constraint violations
- count: use of row-count returned by the `executeUpdate()` method call to check that the update was successful
- param: like count but using a `CallableStatement` output parameter.

To define the result check style, use the `check` parameter.

> The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging, so Hibernate can 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 or mapping files as that will override the Hibernate generated static sql.

Overriding SQL statements for secondary tables is also possible using `@org.hibernate.annotations.Table` and the `sqlInsert`, `sqlUpdate`, `sqlDelete` attributes.

Example 514. Overriding SQL statements for secondary tables

JAVA

```
@Entity(name = "Person")
@Table(name = "person")
@SQLInsert(
    sql = "INSERT INTO person (name, id, valid) VALUES (?, ?, true) "
)
@SQLDelete(
    sql = "UPDATE person SET valid = false WHERE id = ? "
)
@SecondaryTable(name = "person_details",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "person_id"))
@org.hibernate.annotations.Table(
    appliesTo = "person_details",
    sqlInsert = @SQLInsert(
        sql = "INSERT INTO person_details (image, person_id, valid) VALUES (?, ?, true) ",
        check = ResultCheckStyle.COUNT
    ),
    sqlDelete = @SQLDelete(
        sql = "UPDATE person_details SET valid = false WHERE person_id = ? "
    )
)
@Loader(namedQuery = "find_valid_person")
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "find_valid_person",
        query = "select " +
                "    p.id, " +
                "    p.name, " +
                "    pd.image  " +
                "from person p  " +
                "left outer join person_details pd on p.id = pd.person_id  " +
                "where p.id = ? and p.valid = true and pd.valid = true",
        resultClass = Person.class
    )
})
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Column(name = "image", table = "person_details")
    private byte[] image;

    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }
```

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

    public byte[] getImage() {
        return image;
    }

    public void setImage(byte[] image) {
        this.image = image;
    }
}
```

> The SQL is directly executed in your database, so you can use any dialect you like. This will, however, reduce the portability of your mapping if you use database specific SQL.

You can also use stored procedures for customizing the CRUD statements.

Assuming the following stored procedure:

Example 515. Oracle stored procedure to soft-delete a given entity

JAVA

```
statement.executeUpdate(
    "CREATE OR REPLACE PROCEDURE sp_delete_person ( " +
    "   personId IN NUMBER ) " +
    "AS  " +
    "BEGIN " +
    "    UPDATE person SET valid = 0 WHERE id = personId; " +
    "END;"
);}
```

The entity can use this stored procedure to soft-delete the entity in question:

Example 516. Customizing the entity delete statement to use the Oracle stored procedure= instead

```
@SQLDelete(
    sql =   "{ call sp_delete_person( ? ) } ",
    callable = true
)
```
JAVA

> You need to set the `callable` attribute when using a stored procedure instead of an SQL statement.

18. Spatial

18.1. Overview

Hibernate Spatial was originally developed as a generic extension to Hibernate for handling geographic data. Since 5.0, Hibernate Spatial is now part of the Hibernate ORM project, and it allows you to deal with geographic data in a standardized way.

Hibernate Spatial provides a standardized, cross-database interface to geographic data storage and query functions. It supports most of the functions described by the OGC Simple Feature Specification. Supported databases are: Oracle 10g/11g, PostgreSql/PostGIS, MySQL, Microsoft SQL Server and H2/GeoDB.

Spatial data types are not part of the Java standard library, and they are absent from the JDBC specification. Over the years JTS (http://tsusiatsoftware.net/jts/main.html) has emerged the *de facto* standard to fill this gap. JTS is an implementation of the Simple Feature Specification (SFS) (https://portal.opengeospatial.org/files/?artifact_id=829). Many databases on the other hand implement the SQL/MM - Part 3: Spatial Data specification - a related, but broader specification. The biggest difference is that SFS is limited to 2D geometries in the projected plane (although JTS supports 3D coordinates), whereas SQL/MM supports 2-, 3- or 4-dimensional coordinate spaces.

Hibernate Spatial supports two different geometry models: JTS (http://tsusiatsoftware.net/jts/main.html) and geolatte-geom (https://github.com/GeoLatte/geolatte-geom). As already mentioned, JTS is the *de facto* standard. Geolatte-geom (also written by the lead developer of Hibernate Spatial) is a more recent library that supports many features specified in SQL/MM but not available in JTS

(such as support for 4D geometries, and support for extended WKT/WKB formats). Geolatte-geom also implements encoders/decoders for the database native types. Geolatte-geom has good interoperability with JTS. Converting a Geolatte `geometry` to a JTS `geometry, for instance, doesn't require copying of the coordinates. It also delegates spatial processing to JTS.

Whether you use JTS or Geolatte-geom, Hibernate spatial maps the database spatial types to your geometry model of choice. It will, however, always use Geolatte-geom to decode the database native types.

Hibernate Spatial also makes a number of spatial functions available in HQL and in the Criteria Query API. These functions are specified in both SQL/MM as SFS, and are commonly implemented in databases with spatial support (see Hibernate Spatial dialect function support)

18.2. Configuration

Hibernate Spatial requires some configuration prior to start using it.

18.2.1. Dependency

You need to include the `hibernate-spatial` dependency in your build environment. For Maven, you need to add the following dependency:

Example 517. Maven dependency

```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-spatial</artifactId>
    <version>${hibernate.version}</version>
</dependency>
```

18.2.2. Dialects

Hibernate Spatial extends the Hibernate ORM dialects so that the spatial functions of the database are made available within HQL and JPQL. So, for instance, instead of using the `PostgreSQL82Dialect`, we use the Hibernate Spatial extension of that dialect which is the `PostgisDialect`.

Example 518. Specifying a spatial dialect

```xml
<property
    name="hibernate.dialect"
    value="org.hibernate.spatial.dialect.postgis.PostgisDialect"
/>
```

Not all databases support all the functions defined by Hibernate Spatial. The table below provides an overview of the functions provided by each database. If the function is defined in the Simple Feature Specification (https://portal.opengeospatial.org/files/?artifact_id=829), the description references the relevant section.

Table 9. Hibernate Spatial dialect function support

Function	Description	PostgresSQL	Oracle 10g/11g	MySQL	SQLServer	GeoDB (H2)
Basic functions on Geometry						
`int dimension(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`String geometrytype(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`int srid(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`Geometry envelope(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`String astext(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`byte[] asbinary(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`boolean isempty(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`boolean issimple(Geometry)`	SFS §2.1.1.1	✔	✔	✔	✔	✔
`Geometry boundary(Geometry)`	SFS §2.1.1.1	✔	✔	✘	✔	✔
Functions for testing Spatial Relations between geometric objects						
`boolean equals(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean disjoint(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔

`boolean intersects(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean touches(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean crosses(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean within(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean contains(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean overlaps(Geometry, Geometry)`	SFS §2.1.1.2	✔	✔	✔	✔	✔
`boolean relate(Geometry, Geometry, String)`	SFS §2.1.1.2	✔	✔	✘	✔	✔
Functions that support Spatial Analysis						
`double distance(Geometry, Geometry)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
`Geometry buffer(Geometry, double)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
`Geometry convexhull(Geometry)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
`Geometry intersection(Geometry, Geometry)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
`Geometry geomunion(Geometry, Geometry)`	SFS §2.1.1.3 (renamed from union)	✔	✔	✘	✔	✔

`Geometry difference(Geometry, Geometry)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
`Geometry symdifference(Geometry, Geometry)`	SFS §2.1.1.3	✔	✔	✘	✔	✔
Common non-SFS functions						
`boolean dwithin(Geometry, Geometry, double)`	Returns true if the geometries are within the specified distance of one another	✔	✔	✘	✘	✔
`Geometry transform(Geometry, int)`	Returns a new geometry with its coordinates transformed to the SRID referenced by the integer parameter	✔	✔	✘	✘	✘
Spatial aggregate Functions						
`Geometry extent(Geometry)`	Returns a bounding box that bounds the set of returned geometries	✔	✔	✘	✘	✘

Postgis

For Postgis from versions 1.3 and later, the best dialect to use is `org.hibernate.spatial.dialect.postgis.PostgisDialect`.

This translates the HQL spatial functions to the Postgis SQL/MM-compliant functions. For older, pre v1.3 versions of Postgis, which are not SQL/MM compliant, the dialect `org.hibernate.spatial.dialect.postgis.PostgisNoSQLMM` is provided.

MySQL

There are several dialects for MySQL:

`MySQLSpatialDialect`
: a spatially-extended version of Hibernate `MySQLDialect`

`MySQLSpatialInnoDBDialect`
: a spatially-extended version of Hibernate `MySQLInnoDBDialect`

`MySQLSpatial56Dialect`

a spatially-extended version of Hibernate `MySQL5DBDialect`.

`MySQLSpatial5InnoDBDialect`

the same as `MySQLSpatial56Dialect`, but with support for the InnoDB storage engine.

> MySQL versions before 5.6.1 had only limited support for spatial operators. Most operators only took account of the minimum bounding rectangles (MBR) of the geometries, and not the geometries themselves.
>
> This changed in version 5.6.1 were MySQL introduced `ST_*` spatial operators. The dialects `MySQLSpatial56Dialect` and `MySQLSpatial5InnoDBDialect` use these newer, more precise operators.
>
> These dialects may therefore produce results that differ from that of the other spatial dialects.
>
> For more information, see this page in the MySQL reference guide (esp. the section Functions That Test Spatial Relations Between Geometry Objects (https://dev.mysql.com/doc/refman/5.7/en/spatial-relation-functions.html))

Oracle10g/11g

There is currently only one Oracle spatial dialect: `OracleSpatial10gDialect` which extends the Hibernate dialect `Oracle10gDialect`. This dialect has been tested on both Oracle 10g and Oracle 11g with the `SDO_GEOMETRY` spatial database type.

This dialect is the only dialect that can be configured using these Hibernate properties:

`hibernate.spatial.connection_finder`

the fully-qualified classname for the Connection finder for this Dialect (see below).

> *The* `ConnectionFinder` *interface*
>
> The `SDOGeometryType` requires access to an `OracleConnection` object wehen converting a geometry to SDO_GEOMETRY. In some environments, however, the `OracleConnection` is not available (e.g. because a Java EE container or connection pool proxy wraps the connection object in its own `Connection` implementation). A `ConnectionFinder` knows how to retrieve the `OracleConnection` from the wrapper or proxy Connection object that is passed into prepared statements.
>
> The default implementation will, when the passed object is not already an `OracleConnection`, attempt to retrieve the `OracleConnection` by recursive reflection. It will search for methods that return `Connection` objects, execute these methods and check the result. If the result is of type `OracleConnection` the object is returned, otherwise it recurses on it.
>
> In may cases this strategy will suffice. If not, you can provide your own implementation of this interface on the class path, and configure it in the `hibernate.spatial.connection_finder` property. Note that implementations must be thread-safe and have a default no-args constructor.

SQL Server

The dialect `SqlServer2008Dialect` supports the `GEOMETRY` type in SQL Server 2008 and later.

> The `GEOGRAPHY` type is not currently supported.

GeoDB (H2)

The `GeoDBDialect` supports the GeoDB a spatial extension of the H2 in-memory database.

> The dialect has been tested with GeoDB version 0.7

18.3. Types

Hibernate Spatial comes with the following types:

jts_geometry

Handled by `org.hibernate.spatial.JTSGeometryType` it maps a database geometry column type to a `com.vividsolutions.jts.geom.Geometry` entity property type.

geolatte_geometry

Handled by `org.hibernate.spatial.GeolatteGeometryType`, it maps a database geometry column type to an `org.geolatte.geom.Geometry` entity property type.

It suffices to declare a property as either a JTS or an Geolatte-geom `Geometry` and Hibernate Spatial will map it using the relevant type.

Here is an example using JTS:

Example 519. Type mapping

JAVA

```
import com.vividsolutions.jts.geom.Point;

@Entity(name = "Event")
public static class Event {

    @Id
    private Long id;

    private String name;

    private Point location;

    //Getters and setters are omitted for brevity
}
```

We can now treat spatial geometries like any other type.

Example 520. Creating a Point

```java
Event event = new Event();
event.setId( 1L);
event.setName( "Hibernate ORM presentation");
Point point = geometryFactory.createPoint( new Coordinate( 10, 5 ) );
event.setLocation( point );

entityManager.persist( event );
```

Spatial Dialects defines many query functions that are available both in HQL and JPQL queries. Below we show how we could use the `within` function to find all objects within a given spatial extent or window.

Example 521. Querying the geometry

```sql
Polygon window = geometryFactory.createPolygon( coordinates );
Event event = entityManager.createQuery(
    "select e " +
    "from Event e " +
    "where within(e.location, :window) = true", Event.class)
.setParameter("window", window)
.getSingleResult();
```

19. Multitenancy

19.1. What is multitenancy?

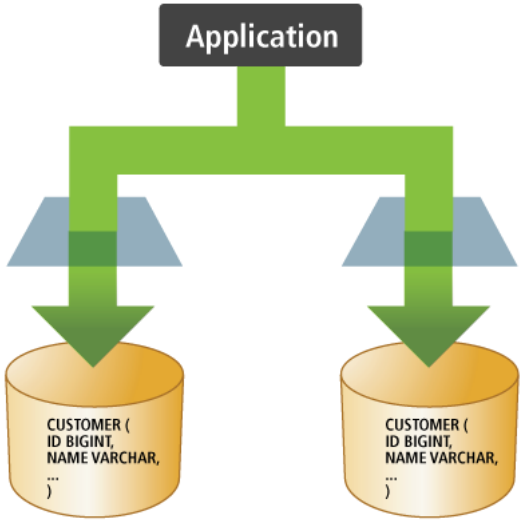
The term multitenancy, in general, is applied to software development to indicate an architecture in which a single running instance of an application simultaneously serves multiple clients (tenants). This is highly common in SaaS solutions. Isolating information (data, customizations, etc.) pertaining to the various tenants is a particular challenge in these systems. This includes the data owned by each tenant stored in the database. It is this last piece, sometimes called multitenant data, on which we will focus.

19.2. Multitenant data approaches

There are three main approaches to isolating information in these multitenant systems which go hand-in-hand with different database schema definitions and JDBC setups.

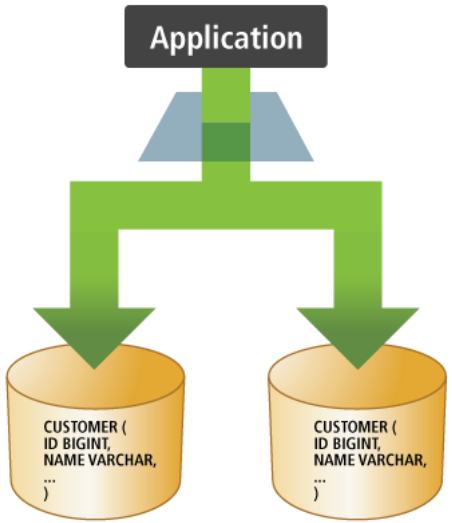
> Each approach has pros and cons as well as specific techniques and considerations. Such topics are beyond the scope of this documentation. Many resources exist which delve into these other topics, like this one (http://msdn.microsoft.com/en-us/library/aa479086.aspx) which does a great job of covering these topics.

19.2.1. Separate database



Each tenant's data is kept in a physically separate database instance. JDBC Connections would point specifically to each database so any pooling would be per-tenant. A general application approach, here, would be to define a JDBC Connection pool per-tenant and to select the pool to use based on the *tenant identifier* associated with the currently logged in user.

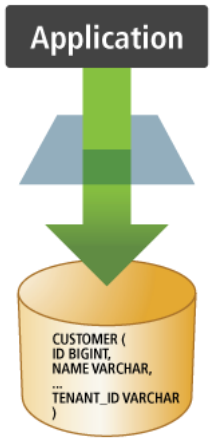
19.2.2. Separate schema



Each tenant's data is kept in a distinct database schema on a single database instance. There are two different ways to define JDBC Connections here:

- Connections could point specifically to each schema as we saw with the `Separate database` approach. This is an option provided that the driver supports naming the default schema in the connection URL or if the pooling mechanism supports naming a schema to use for its Connections. Using this approach, we would have a distinct JDBC Connection pool per-tenant where the pool to use would be selected based on the "tenant identifier" associated with the currently logged in user.
- Connections could point to the database itself (using some default schema) but the Connections would be altered using the SQL `SET SCHEMA` (or similar) command. Using this approach, we would have a single JDBC Connection pool for use to service all tenants, but before using the Connection, it would be altered to reference the schema named by the "tenant identifier" associated with the currently logged in user.

19.3. Partitioned (discriminator) data



All data is kept in a single database schema. The data for each tenant is partitioned by the use of partition value or discriminator. The complexity of this discriminator might range from a simple column value to a complex SQL formula. Again, this approach would use a single Connection pool to service all tenants. However, in this approach, the application needs to alter each and every SQL statement sent to the database to reference the "tenant identifier" discriminator.

19.4. Multitenancy in Hibernate

Using Hibernate with multitenant data comes down to both an API and then integration piece(s). As usual, Hibernate strives to keep the API simple and isolated from any underlying integration complexities. The API is really just defined by passing the tenant identifier as part of opening any session.

Example 522. Specifying tenant identifier from `SessionFactory`

```java
private void doInSession(String tenant, Consumer<Session> function) {
    Session session = null;
    Transaction txn = null;
    try {
        session = sessionFactory
            .withOptions()
            .tenantIdentifier( tenant )
            .openSession();
        txn = session.getTransaction();
        txn.begin();
        function.accept(session);
        txn.commit();
    } catch (Throwable e) {
        if ( txn != null ) txn.rollback();
        throw e;
    } finally {
        if (session != null) {
            session.close();
        }
    }
}
```

Additionally, when specifying the configuration, an `org.hibernate.MultiTenancyStrategy` should be named using the `hibernate.multiTenancy` setting. Hibernate will perform validations based on the type of strategy you specify. The strategy here correlates to the isolation approach discussed above.

NONE

(the default) No multitenancy is expected. In fact, it is considered an error if a tenant identifier is specified when opening a session using this strategy.

SCHEMA

Correlates to the separate schema approach. It is an error to attempt to open a session without a tenant identifier using this strategy. Additionally, a `MultiTenantConnectionProvider` must be specified.

DATABASE

Correlates to the separate database approach. It is an error to attempt to open a session without a tenant identifier using this strategy. Additionally, a `MultiTenantConnectionProvider` must be specified.

DISCRIMINATOR

Correlates to the partitioned (discriminator) approach. It is an error to attempt to open a session without a tenant identifier using this strategy. This strategy is not yet implemented and you can follow its progress via the HHH-6054 Jira issue (https://hibernate.atlassian.net/browse/HHH-6054).

19.4.1. MultiTenantConnectionProvider

When using either the DATABASE or SCHEMA approach, Hibernate needs to be able to obtain Connections in a tenant-specific manner.

That is the role of the `MultiTenantConnectionProvider` contract. Application developers will need to provide an implementation of this contract.

Most of its methods are extremely self-explanatory. The only ones which might not be are `getAnyConnection` and `releaseAnyConnection`. It is important to note also that these methods do not accept the tenant identifier. Hibernate uses these methods during startup to perform various configuration, mainly via the `java.sql.DatabaseMetaData` object.

The `MultiTenantConnectionProvider` to use can be specified in a number of ways:

- Use the `hibernate.multi_tenant_connection_provider` setting. It could name a `MultiTenantConnectionProvider` instance, a `MultiTenantConnectionProvider` implementation class reference or a `MultiTenantConnectionProvider` implementation class name.
- Passed directly to the `org.hibernate.boot.registry.StandardServiceRegistryBuilder`.
- If none of the above options match, but the settings do specify a `hibernate.connection.datasource` value, Hibernate will assume it should use the specific `DataSourceBasedMultiTenantConnectionProviderImpl` implementation which works on a number of pretty reasonable assumptions when running inside of an app server and using one `javax.sql.DataSource` per tenant. See its Javadocs (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/jdbc/connections/spi/DataSourceBasedMultiTenantConnectionProviderImpl.html) for more details.

The following example portrays a `MultiTenantConnectionProvider` implementation that handles multiple `ConnectionProviders`.

Example 523. A `MultiTenantConnectionProvider` implementation

```java
public class ConfigurableMultiTenantConnectionProvider
        extends AbstractMultiTenantConnectionProvider {

    private final Map<String, ConnectionProvider> connectionProviderMap =
        new HashMap<>(  );

    public ConfigurableMultiTenantConnectionProvider(
            Map<String, ConnectionProvider> connectionProviderMap) {
        this.connectionProviderMap.putAll( connectionProviderMap );
    }

    @Override
    protected ConnectionProvider getAnyConnectionProvider() {
        return connectionProviderMap.values().iterator().next();
    }

    @Override
    protected ConnectionProvider selectConnectionProvider(String tenantIdentifier) {
        return connectionProviderMap.get( tenantIdentifier );
    }
}
```

The `ConfigurableMultiTenantConnectionProvider` can be set up as follows:

Example 524. A `MultiTenantConnectionProvider` implementation

```java
private void init() {
    registerConnectionProvider( FRONT_END_TENANT );
    registerConnectionProvider( BACK_END_TENANT );

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

    settings.put( AvailableSettings.MULTI_TENANT, multiTenancyStrategy() );
    settings.put( AvailableSettings.MULTI_TENANT_CONNECTION_PROVIDER,
        new ConfigurableMultiTenantConnectionProvider( connectionProviderMap ) );

    sessionFactory = sessionFactory(settings);
}

protected void registerConnectionProvider(String tenantIdentifier) {
    Properties properties = properties();
    properties.put( Environment.URL,
        tenantUrl(properties.getProperty( Environment.URL ), tenantIdentifier) );

    DriverManagerConnectionProviderImpl connectionProvider =
        new DriverManagerConnectionProviderImpl();
    connectionProvider.configure( properties );
    connectionProviderMap.put( tenantIdentifier, connectionProvider );
}
```

When using multitenancy, it's possible to save an entity with the same identifier across different tenants:

Example 525. A `MultiTenantConnectionProvider` implementation

```
JAVA
doInSession( FRONT_END_TENANT, session -> {
    Person person = new Person(  );
    person.setId( 1L );
    person.setName( "John Doe" );
    session.persist( person );
} );

doInSession( BACK_END_TENANT, session -> {
    Person person = new Person(  );
    person.setId( 1L );
    person.setName( "John Doe" );
    session.persist( person );
} );
```

19.4.2. CurrentTenantIdentifierResolver

`org.hibernate.context.spi.CurrentTenantIdentifierResolver` is a contract for Hibernate to be able to resolve what the application considers the current tenant identifier. The implementation to use is either passed directly to `Configuration` via its `setCurrentTenantIdentifierResolver` method. It can also be specified via the `hibernate.tenant_identifier_resolver` setting.

There are two situations where CurrentTenantIdentifierResolver is used:

- The first situation is when the application is using the `org.hibernate.context.spi.CurrentSessionContext` feature in conjunction with multitenancy. In the case of the current-session feature, Hibernate will need to open a session if it cannot find an existing one in scope. However, when a session is opened in a multitenant environment, the tenant identifier has to be specified. This is where the `CurrentTenantIdentifierResolver` comes into play; Hibernate will consult the implementation you provide to determine the tenant identifier to use when opening the session. In this case, it is required that a `CurrentTenantIdentifierResolver` is supplied.
- The other situation is when you do not want to have to explicitly specify the tenant identifier all the time. If a `CurrentTenantIdentifierResolver` has been specified, Hibernate will use it to determine the default tenant identifier to use when opening the session.

Additionally, if the `CurrentTenantIdentifierResolver` implementation returns `true` for its `validateExistingCurrentSessions` method, Hibernate will make sure any existing sessions that are found in scope have a matching tenant identifier. This capability is only pertinent when the `CurrentTenantIdentifierResolver` is used in current-session settings.

19.4.3. Caching

Multitenancy support in Hibernate works seamlessly with the Hibernate second level cache. The key used to cache data encodes the tenant identifier.

> Currently, schema export will not really work with multitenancy. That may not change.
>
> The JPA expert group is in the process of defining multitenancy support for an upcoming version of the specification.

20. OSGi

20.1. OSGi Specification and Environment

Hibernate targets the OSGi 4.3 spec or later. It was necessary to start with 4.3, over 4.2, due to our dependency on OSGi's `BundleWiring` for entity/mapping scanning.

Hibernate supports three types of configurations within OSGi.

1. Container-Managed JPA Container-Managed JPA
2. Unmanaged JPA Unmanaged JPA
3. Unmanaged Native Unmanaged Native

20.2. hibernate-osgi

Rather than embed OSGi capabilities into hibernate-core, and sub-modules, hibernate-osgi was created. It's purposefully separated, isolating all OSGi dependencies. It provides an OSGi-specific `ClassLoader` (aggregates the container's `ClassLoader` with core and `EntityManager ClassLoader`s), JPA `persistence provider, `SessionFactory` / `EntityManagerFactory` bootstrapping, entities/mappings scanner, and service management.

20.3. features.xml

Apache Karaf environments tend to make heavy use of its "features" concept, where a feature is a set of order-specific bundles focused on a concise capability. These features are typically defined in a `features.xml` file. Hibernate produces and releases its own `features.xml` that defines a core `hibernate-orm`, as well as additional features for optional functionality (caching,

Envers, etc.). This is included in the binary distribution, as well as deployed to the JBoss Nexus repository (using the `org.hibernate` groupId and `hibernate-osgi` with the `karaf.xml` classifier).

Note that our features are versioned using the same ORM artifact versions they wrap. Also, note that the features are heavily tested against Karaf 3.0.3 as a part of our PaxExam-based integration tests. However, they'll likely work on other versions as well.

hibernate-osgi, theoretically, supports a variety of OSGi containers, such as Equinox. In that case, please use `features.xm`l as a reference for necessary bundles to activate and their correct ordering. However, note that Karaf starts a number of bundles automatically, several of which would need to be installed manually on alternatives.

20.4. QuickStarts/Demos

All three configurations have a QuickStart/Demo available in the hibernate-demos (https://github.com/hibernate/hibernate-demos) project:

20.5. Container-Managed JPA

The Enterprise OSGi specification includes container-managed JPA. The container is responsible for discovering persistence units in bundles and automatically creating the `EntityManagerFactory` (one `EntityManagerFactory` per `PersistenceUnit`). It uses the JPA provider (hibernate-osgi) that has registered itself with the OSGi `PersistenceProvider` service.

20.6. Enterprise OSGi JPA Container

In order to utilize container-managed JPA, an Enterprise OSGi JPA container must be active in the runtime. In Karaf, this means Aries JPA, which is included out-of-the-box (simply activate the `jpa` and `transaction` features). Originally, we intended to include those dependencies within our own `features.xml`. However, after guidance from the Karaf and Aries teams, it was pulled out. This allows Hibernate OSGi to be portable and not be directly tied to Aries versions, instead having the user choose which to use.

That being said, the QuickStart/Demo projects include a sample features.xml (https://github.com/hibernate/hibernate-demos/tree/master/hibernate-orm/osgi/managed-jpa/features.xml) showing which features need activated in Karaf in order to support this environment. As mentioned, use this purely as a reference!

20.7. persistence.xml

Similar to any other JPA setup, your bundle must include a `persistence.xml` file. This is typically located in `META-INF`.

20.8. DataSource

Typical Enterprise OSGi JPA usage includes a `DataSource` installed in the container. Your bundle's `persistence.xml` calls out the `DataSource` through JNDI. For example, you could install the following H2 `DataSource`. You can deploy the `DataSource` manually (Karaf has a `deploy` dir), or through a "blueprint bundle" (`blueprint:file:/[PATH]/datasource-h2.xml`).

Example 526. datasource-h2.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
First install the H2 driver using:
> install -s mvn:com.h2database/h2/1.3.163

Then copy this file to the deploy folder
-->
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <bean id="dataSource" class="org.h2.jdbcx.JdbcDataSource">
        <property name="URL" value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE"/>
        <property name="user" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <service interface="javax.sql.DataSource" ref="dataSource">
        <service-properties>
            <entry key="osgi.jndi.service.name" value="jdbc/h2ds"/>
        </service-properties>
    </service>
</blueprint>
```

That `DataSource` is then used by your `persistence.xml` persistence-unit. The following works in Karaf, but the names may need tweaked in alternative containers.

Example 527. META-INF/persistence.xml

```xml
<jta-data-source>osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/h2ds)</jta-data-source>
```

20.9. Bundle Package Imports

Your bundle's manifest will need to import, at a minimum,

- `javax.persistence`
- `org.hibernate.proxy` and `javassist.util.proxy`, due to Hibernate's ability to return proxies for lazy initialization (Javassist enhancement occurs on the entity's `ClassLoader` during runtime).

20.10. Obtaining an EntityManger

The easiest, and most supported, method of obtaining an `EntityManager` utilizes OSGi's `OSGI-INF/blueprint/blueprint.xml` in your bundle. The container takes the name of your persistence unit, then automatically injects an `EntityManager` instance into your given bean attribute.

Example 528. OSGI-INF/blueprint/blueprint.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns:jpa="http://aries.apache.org/xmlns/jpa/v1.0.0"
           xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.0.0"
           default-activation="eager"
           xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <!-- This gets the container-managed EntityManager and injects it into the DataPointServiceImpl bean.
    Assumes DataPointServiceImpl has an "entityManager" field with a getter and setter. -->
    <bean id="dpService" class="org.hibernate.osgitest.DataPointServiceImpl">
        <jpa:context unitname="managed-jpa" property="entityManager"/>
        <tx:transaction method="*" value="Required"/>
    </bean>

    <service ref="dpService" interface="org.hibernate.osgitest.DataPointService"/>

</blueprint>
```

20.11. Unmanaged JPA

Hibernate also supports the use of JPA, unmanaged by the OSGi container. The client bundle is responsible for managing the `EntityManagerFactory` and `EntityManager`s.

20.12. persistence.xml

Similar to any other JPA setup, your bundle must include a `persistence.xml` file. This is typically located in `META-INF`.

20.13. Bundle Package Imports

Your bundle's manifest will need to import, at a minimum,

- javax.persistence
- `org.hibernate.proxy` and `javassist.util.proxy`, due to Hibernate's ability to return proxies for lazy initialization (Javassist enhancement occurs on the entity's `ClassLoader` during runtime)
- JDBC driver package (example: `org.h2`)
- `org.osgi.framework`, necessary to discover the `EntityManagerFactory` (described below)

20.14. Obtaining an EntityMangerFactory

`hibernate-osgi` registers an OSGi service, using the JPA `PersistenceProvider` interface name, that bootstraps and creates an `EntityManagerFactory` specific for OSGi environments.

> It is VITAL that your `EntityManagerFactory` be obtained through the service, rather than creating it manually. The service handles the OSGi `ClassLoader`, discovered extension points, scanning, etc. Manually creating an `EntityManagerFactory` is guaranteed to NOT work during runtime!

Example 529. Discover/Use `EntityManagerFactory`

JAVA

```
public class HibernateUtil {

    private EntityManagerFactory emf;

    public EntityManager getEntityManager() {
        return getEntityManagerFactory().createEntityManager();
    }

    private EntityManagerFactory getEntityManagerFactory() {
        if ( emf == null ) {
            Bundle thisBundle = FrameworkUtil.getBundle( HibernateUtil.class );
            BundleContext context = thisBundle.getBundleContext();

            ServiceReference serviceReference = context.getServiceReference( PersistenceProvider.class.getName() );
            PersistenceProvider persistenceProvider = ( PersistenceProvider ) context.getService( serviceReference
);

            emf = persistenceProvider.createEntityManagerFactory( "YourPersistenceUnitName", null );
        }
        return emf;
    }
}
```

20.15. Unmanaged Native

Native Hibernate use is also supported. The client bundle is responsible for managing the `SessionFactory` and `Session`s.

20.16. Bundle Package Imports

Your bundle's manifest will need to import, at a minimum,

- javax.persistence
- `org.hibernate.proxy` and `javassist.util.proxy`, due to Hibernate's ability to return proxies for lazy initialization (Javassist enhancement occurs on the entity's `ClassLoader` during runtime)
- JDBC driver package (example: `org.h2`)

- `org.osgi.framework`, necessary to discover the `SessionFactory` (described below)
- `org.hibernate.*` packages, as necessary (ex: cfg, criterion, service, etc.)

20.17. Obtaining an SessionFactory

`hibernate-osgi` registers an OSGi service, using the `SessionFactory` interface name, that bootstraps and creates a `SessionFactory` specific for OSGi environments.

> It is VITAL that your `SessionFactory` be obtained through the service, rather than creating it manually. The service handles the OSGi `ClassLoader`, discovered extension points, scanning, etc. Manually creating a `SessionFactory` is guaranteed to NOT work during runtime!

Example 530. Discover/Use `SessionFactory`

```
JAVA
public class HibernateUtil {

    private SessionFactory sf;

    public Session getSession() {
        return getSessionFactory().openSession();
    }

    private SessionFactory getSessionFactory() {
        if ( sf == null ) {
            Bundle thisBundle = FrameworkUtil.getBundle( HibernateUtil.class );
            BundleContext context = thisBundle.getBundleContext();

            ServiceReference sr = context.getServiceReference( SessionFactory.class.getName() );
            sf = ( SessionFactory ) context.getService( sr );
        }
        return sf;
    }
}
```

20.18. Optional Modules

The unmanaged-native (https://github.com/hibernate/hibernate-demos/tree/master/hibernate-orm/osgi/unmanaged-native) demo project displays the use of optional Hibernate modules. Each module adds additional dependency bundles that must first be activated, either manually or through an additional feature. As of ORM 4.2, Envers is fully supported. Support for C3P0, Proxool, EhCache, and Infinispan were added in 4.3, however none of their 3rd party libraries currently work in OSGi (lots of `ClassLoader` problems, etc.). We're tracking the issues in JIRA.

20.19. Extension Points

Multiple contracts exist to allow applications to integrate with and extend Hibernate capabilities. Most apps utilize JDK services to provide their implementations. `hibernate-osgi` supports the same extensions through OSGi services. Implement and register them in any of the three configurations. `hibernate-osgi` will discover and integrate them during `EntityManagerFactory` / `SessionFactory` bootstrapping. Supported extension points are as follows. The specified interface should be used during service registration.

`org.hibernate.integrator.spi.Integrator`
(as of 4.2)

`org.hibernate.boot.registry.selector.StrategyRegistrationProvider`
(as of 4.3)

`org.hibernate.boot.model.TypeContributor`
(as of 4.3)

JTA's `javax.transaction.TransactionManager` and `javax.transaction.UserTransaction` (as of 4.2), however these are typically provided by the OSGi container.

The easiest way to register extension point implementations is through a `blueprint.xml` file. Add `OSGI-INF/blueprint/blueprint.xml` to your classpath. Envers' blueprint is a great example:

Example 531. Example extension point registrations in blueprint.xml

```xml
<!--
  ~ Hibernate, Relational Persistence for Idiomatic Java
  ~
  ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.
  ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
  -->
<blueprint default-activation="eager"
           xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <bean id="integrator" class="org.hibernate.envers.boot.internal.EnversIntegrator"/>
    <service ref="integrator" interface="org.hibernate.integrator.spi.Integrator"/>

    <bean id="typeContributor"
          class="org.hibernate.envers.boot.internal.TypeContributorImpl"/>
    <service ref="typeContributor" interface="org.hibernate.boot.model.TypeContributor"/>

</blueprint>
```

Extension points can also be registered programmatically with `BundleContext#registerService`, typically within your `BundleActivator#start`.

20.20. Caveats

- Technically, multiple persistence units are supported by Enterprise OSGi JPA and unmanaged Hibernate JPA use. However, we cannot currently support this in OSGi. In Hibernate 4, only one instance of the OSGi-specific `ClassLoader` is used per Hibernate bundle, mainly due to heavy use of static TCCL utilities. We hope to support one OSGi `ClassLoader` per persistence unit in Hibernate 5.
- Scanning is supported to find non-explicitly listed entities and mappings. However, they MUST be in the same bundle as your persistence unit (fairly typical anyway). Our OSGi `ClassLoader` only considers the "requesting bundle" (hence the requirement on using services to create `EntityManagerFactory` / `SessionFactory`), rather than attempting to scan all available bundles. This is primarily for versioning considerations, collision protections, etc.
- Some containers (ex: Aries) always return true for `PersistenceUnitInfo#excludeUnlistedClasses`, even if your `persistence.xml` explicitly has `exclude-unlisted-classes` set to `false`. They claim it's to protect JPA providers from having to implement scanning ("we handle it for you"), even though we still want to support it in many cases. The work around is to set `hibernate.archive.autodetection` to, for example, `hbm,class`. This tells hibernate to ignore the `excludeUnlistedClasses` value and scan for `*.hbm.xml` and entities regardless.
- Scanning does not currently support annotated packages on `package-info.java`.
- Currently, Hibernate OSGi is primarily tested using Apache Karaf and Apache Aries JPA. Additional testing is needed with Equinox, Gemini, and other container providers.
- Hibernate ORM has many dependencies that do not currently provide OSGi manifests. The QuickStart tutorials make heavy use of 3rd party bundles (SpringSource, ServiceMix) or the `wrap:…` operator.

21. Envers

21.1. Basics

To audit changes that are performed on an entity, you only need two things:

- the `hibernate-envers` jar on the classpath,
- an `@Audited` annotation on the entity.

> Unlike in previous versions, you no longer need to specify listeners in the Hibernate configuration file. Just putting the Envers jar on the classpath is enough because listeners will be registered automatically.

And that's all. You can create, modify and delete the entities as always.

If you look at the generated schema for your entities, or at the data persisted by Hibernate, you will notice that there are no changes. However, for each audited entity, a new table is introduced - `entity_table_AUD`, which stores the historical data, whenever you commit a transaction.

> Envers automatically creates audit tables if `hibernate.hbm2ddl.auto` option is set to `create`, `create-drop` or `update`. Appropriate DDL statements can also be generated with an Ant task in Generating schema with Ant.

Instead of annotating the whole class and auditing all properties, you can annotate only some persistent properties with `@Audited`. This will cause only these properties to be audited.

The audit (history) of an entity can be accessed using the `AuditReader` interface, which can be obtained having an open `EntityManager` or `Session` via the `AuditReaderFactory`. See the [Javadocs] (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/envers/AuditReaderFactory.html) for these classes for details on the functionality offered.

21.2. Configuration

It is possible to configure various aspects of Hibernate Envers behavior, such as table names, etc.

Table 10. Envers Configuration Properties

Property name	Default value	Description
`org.hibernate.envers.audit_table_prefix`		String that will be name of the entity
`org.hibernate.envers.audit_table_suffix`	`_AUD`	String that will be of the entity and t table name Perso table to store hist
`org.hibernate.envers.revision_field_name`	`REV`	Name of a field in
`org.hibernate.envers.revision_type_field_name`	`REVTYPE`	Name of a field in (currently, this ca
`org.hibernate.envers.revision_on_collection_change`	`true`	Should a revision can be either a co attribute in a one-
`org.hibernate.envers.do_not_audit_optimistic_locking_field`	`true`	When true, prope `@Version`, will no normally doesn't
`org.hibernate.envers.store_data_at_delete`	`false`	Should the entity (instead of only st normally needed, however, it is easi data that the entit
`org.hibernate.envers.default_schema`	`null` (same schema as table being audited)	The default schem overridden using the schema will b

Property name	Default value	Description
org.hibernate.envers.default_catalog	null (same catalog as table being audited)	The default catalo using the @AuditT will be the same a
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	The audit strategy stores only the rev org.hibernate.e start revision and was valid, hence t
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	The column name property is only v
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	Should the timest valid, in addition Audit records out Partitioning requi evaluated if the Va
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	Column name of t valid. Only used if org.hibernate.e evaluates to true
org.hibernate.envers.use_revision_entity_with_native_id	true	Boolean flag that implementation o database engine d property to false. org.hibernate. org.hibernate. org.hibernate.
org.hibernate.envers.track_entities_changed_in_revision	false	Should entity type The default imple of modified persis (foreign key to RE Tracking entity na modified in a give

Property name	Default value	Description
org.hibernate.envers.global_with_modified_flag	false , can be individually overridden with @Audited(withModifiedFlag=true)	Should property m properties. When the audit tables w changed in the giv selected entities o information, refe revisions of entity
org.hibernate.envers.modified_flag_suffix	_MOD	The suffix for colu "age", will by defa
org.hibernate.envers.embeddable_set_ordinal_field_name	SETORDINAL	Name of column u elements.
org.hibernate.envers.cascade_delete_revision	false	While deleting rev Requires database
org.hibernate.envers.allow_identifier_reuse	false	Guarantees prope identifiers of dele identifier.

> The following configuration options have been added recently and should be regarded as experimental:
>
> 1. org.hibernate.envers.track_entities_changed_in_revision
> 2. org.hibernate.envers.using_modified_flag
> 3. org.hibernate.envers.modified_flag_suffix

21.3. Additional mapping annotations

The name of the audit table can be set on a per-entity basis, using the @AuditTable annotation. It may be tedious to add this annotation to every audited entity, so if possible, it's better to use a prefix/suffix.

If you have a mapping with secondary tables, audit tables for them will be generated in the same way (by adding the prefix and suffix). If you wish to overwrite this behavior, you can use the `@SecondaryAuditTable` and `@SecondaryAuditTables` annotations.

If you'd like to override auditing behavior of some fields/properties inherited from `@MappedSuperclass` or in an embedded component, you can apply the `@AuditOverride( s )` annotation on the subtype or usage site of the component.

If you want to audit a relation mapped with `@OneToMany` and `@JoinColumn`, please see Mapping exceptions for a description of the additional `@AuditJoinTable` annotation that you'll probably want to use.

If you want to audit a relation, where the target entity is not audited (that is the case for example with dictionary-like entities, which don't change and don't have to be audited), just annotate it with `@Audited( targetAuditMode = RelationTargetAuditMode.NOT_AUDITED )`. Then, while reading historic versions of your entity, the relation will always point to the "current" related entity. By default Envers throws `javax.persistence.EntityNotFoundException` when "current" entity does not exist in the database. Apply `@NotFound( action = NotFoundAction.IGNORE )` annotation to silence the exception and assign null value instead. The hereby solution causes implicit eager loading of to-one relations.

If you'd like to audit properties of a superclass of an entity, which are not explicitly audited (they don't have the `@Audited` annotation on any properties or on the class), you can set the `@AuditOverride( forClass = SomeEntity.class, isAudited = true/false )` annotation.

> The `@Audited` annotation also features an `auditParents` attribute but it's now deprecated in favor of `@AuditOverride`,

21.4. Choosing an audit strategy

After the basic configuration, it is important to choose the audit strategy that will be used to persist and retrieve audit information. There is a trade-off between the performance of persisting and the performance of querying the audit information. Currently, there are two audit strategies.

1. The default audit strategy persists the audit data together with a start revision. For each row inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, together with the start revision of its validity. Rows in the audit tables are never updated after insertion. Queries of audit information use subqueries to select the applicable rows in the audit tables.

These subqueries are notoriously slow and difficult to index.

2. The alternative is a validity audit strategy. This strategy stores the start-revision and the end-revision of audit information. For each row inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, together with the start revision of its validity. But at the same time the end-revision field of the previous audit rows (if available) are set to this revision. Queries on the audit information can then use 'between start and end revision' instead of subqueries as used by the default audit strategy.

```
The consequence of this strategy is that persisting audit information will be a bit slower because of the extra
updates involved,
but retrieving audit information will be a lot faster.
This can be improved even further by adding extra indexes.
```

21.5. Revision Log

When Envers starts a new revision, it creates a new revision entity which stores information about the revision. By default, that includes just:

revision number

An integral value (`int/Integer` or `long/Long`). Essentially the primary key of the revision

revision timestamp

either a `long/Long` or `java.util.Date` value representing the instant at which the revision was made. When using a `java.util.Date` , instead of a `long/Long` for the revision timestamp, take care not to store it to a column data type which will loose precision.

Envers handles this information as an entity. By default it uses its own internal class to act as the entity, mapped to the `REVINFO` table. You can, however, supply your own approach to collecting this information which might be useful to capture additional details such as who made a change or the ip address from which the request came. There are two things you need to make this work:

1. First, you will need to tell Envers about the entity you wish to use. Your entity must use the `@org.hibernate.envers.RevisionEntity` annotation. It must define the two attributes described above annotated with `@org.hibernate.envers.RevisionNumber` and `@org.hibernate.envers.RevisionTimestamp` , respectively. You can extend from `org.hibernate.envers.DefaultRevisionEntity` , if you wish, to inherit all these required behaviors.

```
Simply add the custom revision entity as you do your normal entities and Envers will _find it_.
```

> It is an error for there to be multiple entities marked as `@org.hibernate.envers.RevisionEntity`

2. Second, you need to tell Envers how to create instances of your revision entity which is handled by the `newRevision( Object revisionEntity )` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/envers/RevisionListener.html#newRevision-java.lang.Object-) method of the `org.hibernate.envers.RevisionListener` interface.

```
You tell Envers your custom `org.hibernate.envers.RevisionListener` implementation to use by specifying it on the
`@org.hibernate.envers.RevisionEntity` annotation, using the value attribute.
If your `RevisionListener` class is inaccessible from `@RevisionEntity` (e.g. it exists in a different module), set
`org.hibernate.envers.revision_listener` property to its fully qualified class name.
Class name defined by the configuration parameter overrides revision entity's value attribute.
```

JAVA

```
@RevisionEntity( MyCustomRevisionListener.class )
public class MyCustomRevisionEntity {
    ...
}

public class MyCustomRevisionListener implements RevisionListener {
    public void newRevision( Object revisionEntity ) {
        MyCustomRevisionEntity customRevisionEntity = ( MyCustomRevisionEntity ) revisionEntity;
    }
}
```

Example 532. ExampleRevEntity.java

```java
package `org.hibernate.envers.example;`

import `org.hibernate.envers.RevisionEntity;`
import `org.hibernate.envers.DefaultRevisionEntity;`

import javax.persistence.Entity;

@Entity
@RevisionEntity( ExampleListener.class )
public class ExampleRevEntity extends DefaultRevisionEntity {
    private String username;

    public String getUsername() { return username; }
    public void setUsername( String username ) { this.username = username; }
}
```

Example 533. ExampleListener.java

```java
package `org.hibernate.envers.example;`

import `org.hibernate.envers.RevisionListener;`
import org.jboss.seam.security.Identity;
import org.jboss.seam.Component;

public class ExampleListener implements RevisionListener {

    public void newRevision( Object revisionEntity ) {
        ExampleRevEntity exampleRevEntity = ( ExampleRevEntity ) revisionEntity;
        Identity identity =
            (Identity) Component.getInstance( "org.jboss.seam.security.identity" );

        exampleRevEntity.setUsername( identity.getUsername() );
    }
}
```

> An alternative method to using the `org.hibernate.envers.RevisionListener` is to instead call the `getCurrentRevision( Class<T> revisionEntityClass, boolean persist )` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/envers/AuditReader.html#getCurrentRevision-java.lang.Class-boolean-) method of the `org.hibernate.envers.AuditReader` interface to obtain the current revision, and fill it with desired information. The method accepts a `persist` parameter indicating whether the revision entity should be persisted prior to returning from this method:
>
> `true`
> ensures that the returned entity has access to its identifier value (revision number), but the revision entity will be persisted regardless of whether there are any audited entities changed.
>
> `false`
> means that the revision number will be `null`, but the revision entity will be persisted only if some audited entities have changed.

21.6. Tracking entity names modified during revisions

By default entity types that have been changed in each revision are not being tracked. This implies the necessity to query all tables storing audited data in order to retrieve changes made during specified revision. Envers provides a simple mechanism that creates `REVCHANGES` table which stores entity names of modified persistent objects. Single record encapsulates the revision identifier (foreign key to `REVINFO` table) and a string value.

Tracking of modified entity names can be enabled in three different ways:

1. Set `org.hibernate.envers.track_entities_changed_in_revision` parameter to `true`. In this case `org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity` will be implicitly used as the revision log entity.
2. Create a custom revision entity that extends `org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity` class.

```
JAVA
@RevisionEntity
public class ExtendedRevisionEntity extends DefaultTrackingModifiedEntitiesRevisionEntity {
        ...
}
```

3. Mark an appropriate field of a custom revision entity with `@org.hibernate.envers.ModifiedEntityNames` annotation. The property is required to be of `Set<String>` type.

```
@RevisionEntity
public class AnnotatedTrackingRevisionEntity {
    ...

    @ElementCollection
    @JoinTable( name = "REVCHANGES", joinColumns = @JoinColumn( name = "REV" ) )
    @Column( name = "ENTITYNAME" )
    @ModifiedEntityNames
    private Set<String> modifiedEntityNames;

    ...
}
```
JAVA

Users, that have chosen one of the approaches listed above, can retrieve all entities modified in a specified revision by utilizing API described in Querying for entities modified in a given revision.

Users are also allowed to implement custom mechanism of tracking modified entity types. In this case, they shall pass their own implementation of `org.hibernate.envers.EntityTrackingRevisionListener` interface as the value of `@org.hibernate.envers.RevisionEntity` annotation. `EntityTrackingRevisionListener` interface exposes one method that notifies whenever audited entity instance has been added, modified or removed within current revision boundaries.

Example 534. CustomEntityTrackingRevisionListener.java

```
public class CustomEntityTrackingRevisionListener implements EntityTrackingRevisionListener {

    @Override
    public void entityChanged( Class entityClass, String entityName,
                               Serializable entityId, RevisionType revisionType,
                               Object revisionEntity ) {
        String type = entityClass.getName();
        ( ( CustomTrackingRevisionEntity ) revisionEntity ).addModifiedEntityType( type );
    }

    @Override
    public void newRevision( Object revisionEntity ) {
    }
}
```
JAVA

Example 535. CustomTrackingRevisionEntity.java

```java
@Entity
@RevisionEntity( CustomEntityTrackingRevisionListener.class )
public class CustomTrackingRevisionEntity {

    @Id
    @GeneratedValue
    @RevisionNumber
    private int customId;

    @RevisionTimestamp
    private long customTimestamp;

    @OneToMany( mappedBy="revision", cascade={ CascadeType.PERSIST, CascadeType.REMOVE } )
    private Set<ModifiedEntityTypeEntity> modifiedEntityTypes = new HashSet<ModifiedEntityTypeEntity>();

    public void addModifiedEntityType( String entityClassName ) {
        modifiedEntityTypes.add( new ModifiedEntityTypeEntity( this, entityClassName ) );
    }

    ...
}
```

Example 536. ModifiedEntityTypeEntity.java

```java
@Entity
public class ModifiedEntityTypeEntity {

    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne
    private CustomTrackingRevisionEntity revision;

    private String entityClassName;

    ...
}
```

```java
CustomTrackingRevisionEntity revEntity =
    getAuditReader().findRevision( CustomTrackingRevisionEntity.class, revisionNumber );

Set<ModifiedEntityTypeEntity> modifiedEntityTypes = revEntity.getModifiedEntityTypes();
```

21.7. Tracking entity changes at property level

By default, the only information stored by Envers are revisions of modified entities. This approach lets user create audit queries based on historical values of entity properties. Sometimes it is useful to store additional metadata for each revision, when you are interested also in the type of changes, not only about the resulting values.

The feature described in Tracking entity names modified during revisions makes it possible to tell which entities were modified in a given revision.

The feature described here takes it one step further. "Modification Flags" enable Envers to track which properties of audited entities were modified in a given revision.

Tracking entity changes at property level can be enabled by:

1. setting `org.hibernate.envers.global_with_modified_flag` configuration property to `true`. This global switch will cause adding modification flags to be stored for all audited properties of all audited entities.
2. using `@Audited( withModifiedFlag=true )` on a property or on an entity.

The trade-off coming with this functionality is an increased size of audit tables and a very little, almost negligible, performance drop during audit writes. This is due to the fact that every tracked property has to have an accompanying boolean column in the schema that stores information about the property modifications. Of course it is Envers job to fill these columns accordingly - no additional work by the developer is required. Because of costs mentioned, it is recommended to enable the feature selectively, when needed with use of the granular configuration means described above.

To see how "Modified Flags" can be utilized, check out the very simple query API that uses them: Querying for revisions of entity that modified given property.

21.8. Queries

You can think of historic data as having two dimensions:

horizontal
: is the state of the database at a given revision. Thus, you can query for entities as they were at revision N.

vertical
: are the revisions, at which entities changed. Hence, you can query for revisions, in which a given entity changed.

The queries in Envers are similar to Hibernate Criteria queries, so if you are common with them, using Envers queries will be much easier.

The main limitation of the current queries implementation is that you cannot traverse relations. You can only specify constraints on the ids of the related entities, and only on the "owning" side of the relation. This however will be changed in future releases.

Please note, that queries on the audited data will be in many cases much slower than corresponding queries on "live" data, as they involve correlated subselects.

Queries are improved both in terms of speed and possibilities, when using the valid-time audit strategy, that is when storing both start and end revisions for entities. See Configuration.

21.9. Querying for entities of a class at a given revision

The entry point for this type of queries is:

JAVA
```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision( MyEntity.class, revisionNumber );
```

You can then specify constraints, which should be met by the entities returned, by adding restrictions, which can be obtained using the `AuditEntity` factory class. For example, to select only entities where the "name" property is equal to "John":

JAVA
```
query.add( AuditEntity.property( "name" ).eq(  "John" ) );
```

And to select only entities that are related to a given entity:

JAVA
```
query.add( AuditEntity.property( "address" ).eq( relatedEntityInstance ) );
// or
query.add( AuditEntity.relatedId( "address" ).eq( relatedEntityId ) );
// or
query.add( AuditEntity.relatedId( "address" ).in( relatedEntityId1, relatedEntityId2 ) );
```

You can limit the number of results, order them, and set aggregations and projections (except grouping) in the usual way. When your query is complete, you can obtain the results by calling the `getSingleResult()` or `getResultList()` methods.

A full query, can look for example like this:

JAVA
```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision( Person.class, 12 )
    .addOrder( AuditEntity.property( "surname" ).desc() )
    .add( AuditEntity.relatedId( "address" ).eq( addressId ) )
    .setFirstResult( 4 )
    .setMaxResults( 2 )
    .getResultList();
```

21.10. Querying for revisions, at which entities of a given class changed

The entry point for this type of queries is:

JAVA
```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity( MyEntity.class, false, true );
```

You can add constraints to this query in the same way as to the previous one. There are some additional possibilities:

1. using `AuditEntity.revisionNumber()` you can specify constraints, projections and order on the revision number, in which the audited entity was modified
2. similarly, using `AuditEntity.revisionProperty( propertyName )` you can specify constraints, projections and order on a property of the revision entity, corresponding to the revision in which the audited entity was modified
3. `AuditEntity.revisionType()` gives you access as above to the type of the revision (`ADD` , `MOD` , `DEL`).

Using these methods, you can order the query results by revision number, set projection or constraint the revision number to be greater or less than a specified value, etc. For example, the following query will select the smallest revision number, at which entity of class `MyEntity` with id `entityId` has changed, after revision number 42:

JAVA

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity( MyEntity.class, false, true )
    .setProjection( AuditEntity.revisionNumber().min() )
    .add( AuditEntity.id().eq( entityId ) )
    .add( AuditEntity.revisionNumber().gt( 42 ) )
    .getSingleResult();
```

The second additional feature you can use in queries for revisions is the ability to *maximize*/*minimize* a property. For example, if you want to select the smallest possibler revision at which the value of the `actualDate` for a given entity was larger then a given value:

JAVA

```
Number revision = (Number) getAuditReader().createQuery()
        .forRevisionsOfEntity( MyEntity.class, false, true) // We are only interested in the first revision
        .setProjection( AuditEntity.revisionNumber().min() )
        .add( AuditEntity.property( "actualDate" ).minimize()
        .add( AuditEntity.property( "actualDate" ).ge( givenDate ) )
        .add( AuditEntity.id().eq( givenEntityId ) )) .getSingleResult();
```

The `minimize()` and `maximize()` methods return a criteria, to which you can add constraints, which must be met by the entities with the *maximized*/*minimized* properties.

> `AggregatedAuditExpression#computeAggregationInInstanceContext()` enables the possibility to compute aggregated expression in the context of each entity instance separately. It turns out useful when querying for latest revisions of all entities of a particular type.

You probably also noticed that there are two boolean parameters, passed when creating the query.

`selectEntitiesOnly`

the first parameter is only valid when you don't set an explicit projection. If true, the result of the query will be a list of entities (which changed at revisions satisfying the specified constraints). If false, the result will be a list of three element arrays:

- the first element will be the changed entity instance.
- the second will be an entity containing revision data (if no custom entity is used, this will be an instance of `DefaultRevisionEntity`).
- the third will be the type of the revision (one of the values of the `RevisionType` enumeration: `ADD`, `MOD`, `DEL`).

`selectDeletedEntities`

the second parameter specifies if revisions, in which the entity was deleted should be included in the results. If yes, such entities will have the revision type `DEL` and all fields, except the id, `null`.

21.11. Querying for revisions of entity that modified given property

For the two types of queries described above it's possible to use special `Audit` criteria called `hasChanged()` and `hasNotChanged()` that makes use of the functionality described in Tracking entity changes at property level. They're best suited for vertical queries, however existing API doesn't restrict their usage for horizontal ones.

Let's have a look at following examples:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
        .forRevisionsOfEntity( MyEntity.class, false, true )
        .add( AuditEntity.id().eq( id ) );
        .add( AuditEntity.property( "actualDate" ).hasChanged() );
```

This query will return all revisions of `MyEntity` with given `id`, where the `actualDate` property has been changed. Using this query we won't get all other revisions in which `actualDate` wasn't touched. Of course, nothing prevents user from combining `hasChanged` condition with some additional criteria - add method can be used here in a normal way.

JAVA

```
AuditQuery query = getAuditReader().createQuery()
        .forEntitiesAtRevision( MyEntity.class, revisionNumber )
        .add( AuditEntity.property( "prop1" ).hasChanged() )
        .add( AuditEntity.property( "prop2" ).hasNotChanged() );
```

This query will return horizontal slice for `MyEntity` at the time `revisionNumber` was generated. It will be limited to revisions that modified `prop1` but not `prop2`.

Note that the result set will usually also contain revisions with numbers lower than the `revisionNumber`, so wem cannot read this query as "Give me all MyEntities changed in `revisionNumber` with `prop1` modified and `prop2` untouched". To get such result we have to use the `forEntitiesModifiedAtRevision` query:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
        .forEntitiesModifiedAtRevision( MyEntity.class, revisionNumber )
        .add( AuditEntity.property( "prop1" ).hasChanged() )
        .add( AuditEntity.property( "prop2" ).hasNotChanged() );
```

21.12. Querying for entities modified in a given revision

The basic query allows retrieving entity names and corresponding Java classes changed in a specified revision:

JAVA

```
modifiedEntityTypes = getAuditReader()
        .getCrossTypeRevisionChangesReader()
        .findEntityTypes( revisionNumber );
```

Other queries (also accessible from `org.hibernate.envers.CrossTypeRevisionChangesReader`):

`List<Object> findEntities( Number )`

Returns snapshots of all audited entities changed (added, updated and removed) in a given revision. Executes `N+1` SQL queries, where `N` is a number of different entity classes modified within specified revision.

`List<Object> findEntities( Number, RevisionType )`

Returns snapshots of all audited entities changed (added, updated or removed) in a given revision filtered by modification type. Executes `N+1` SQL queries, where `N` is a number of different entity classes modified within specified revision.

`Map<RevisionType, List<Object>> findEntitiesGroupByRevisionType( Number )`

Returns a map containing lists of entity snapshots grouped by modification operation (e.g. addition, update and removal). Executes `3N+1` SQL queries, where `N` is a number of different entity classes modified within specified revision.

Note that methods described above can be legally used only when the default mechanism of tracking changed entity names is enabled (see Tracking entity names modified during revisions).

21.13. Querying for entities using entity relation joins

Audit queries support the ability to apply constraints, projections, and sort operations based on entity relations. In order to traverse entity relations through an audit query, you must use the relation traversal API with a join type.

> Relation join queries are considered experimental and may change in future releases.

> Relation joins can only be applied to `*-to-one` mappings and can only be specified using `JoinType.LEFT` or `JoinType.INNER`.

The basis for creating an entity relation join query is as follows:

JAVA

```
// create an inner join query
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.INNER );

// create a left join query
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.LEFT );
```

Like any other query, constraints may be added to restrict the results. For example, to find all `Car` entities that have an owner with a name starting with `Joe`, you would use:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.INNER )
    .add( AuditEntity.property( "name" ).like( "Joe%" ) );
```

It is also possible to traverse beyond the first relation in an entity graph. For example, to find all `Car` entities where the owner's address has a street number that equals `1234`:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.INNER )
    .traverseRelation( "address", JoinType.INNER )
    .add( AuditEntity.property( "streetNumber" ).eq( 1234 ) );
```

Complex constraints may also be added that are applicable to properties of nested relations or the base query entity or relation state, such as testing for `null`. For example, the following query illustrates how to find all `Car` entities where the owner's age is `20` or that the car has *no* owner:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.LEFT, "p" )
    .up()
    .add(
        AuditEntity.or(
            AuditEntity.property( "p", "age" ).eq( 20 ),
            AuditEntity.relatedId( "owner" ).eq( null )
        )
    )
    .addOrder( AuditEntity.property( "make" ).asc() );
```

> Queries can use the `up` method to navigate back up the entity graph.

Disjunction criterion may also be applied to relation join queries. For example, the following query will find all `Car` entities where the owner's age is `20` or that the owner lives at an address where the street number equals `1234`:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.INNER, "p" )
    .traverseRelation( "address", JoinType.INNER, "a" )
    .up()
    .up()
    .add(
        AuditEntity.disjunction()
            .add( AuditEntity.property( "p", "age" ).eq( 20 ) )
            .add( AuditEntity.property( "a", "streetNumber" ).eq( 1234 )
        )
    )
    .addOrder( AuditEntity.property( "make" ).asc() );
```

Lastly, this example illustrates how related entity properties can be compared as a constraint. This query shows how to find the `Car` entities where the owner's `age` equals the `streetNumber` of where the owner lives:

JAVA

```
AuditQuery query = getAuditReader().createQuery()
    .forEntitiesAtRevision( Car.class, 1 )
    .traverseRelation( "owner", JoinType.INNER, "p" )
    .traverseRelation( "address", JoinType.INNER, "a" )
    .up()
    .up()
    .add( AuditEntity.property( "p", "age" ).eqProperty( "a", "streetNumber" ) );
```

21.14. Conditional auditing

Envers persists audit data in reaction to various Hibernate events (e.g. `post update`, `post insert`, and so on), using a series of event listeners from the `org.hibernate.envers.event.spi` package. By default, if the Envers jar is in the classpath, the event listeners are auto-registered with Hibernate.

Conditional auditing can be implemented by overriding some of the Envers event listeners. To use customized Envers event listeners, the following steps are needed:

1. Turn off automatic Envers event listeners registration by setting the `hibernate.listeners.envers.autoRegister` Hibernate property to `false`.
2. Create subclasses for appropriate event listeners. For example, if you want to conditionally audit entity insertions, extend the `org.hibernate.envers.event.spi.EnversPostInsertEventListenerImpl` class. Place the conditional-auditing logic in the subclasses, call the super method if auditing should be performed.
3. Create your own implementation of `org.hibernate.integrator.spi.Integrator`, similar to `org.hibernate.envers.boot.internal.EnversIntegrator`. Use your event listener classes instead of the default ones.
4. For the integrator to be automatically used when Hibernate starts up, you will need to add a `META-INF/services/org.hibernate.integrator.spi.Integrator` file to your jar. The file should contain the fully qualified name of the class implementing the interface.

> The use of `hibernate.listeners.envers.autoRegister` has been deprecated. A new configuration setting `hibernate.envers.autoRegisterListeners` should be used instead.

21.15. Understanding the Envers Schema

For each audited entity (that is, for each entity containing at least one audited field), an audit table is created. By default, the audit table's name is created by adding an "_AUD" suffix to the original table name, but this can be overridden by specifying a different suffix/prefix in the configuration properties or per-entity using the `@org.hibernate.envers.AuditTable` annotation.

The audit table contains the following columns:

id
: `id` of the original entity (this can be more then one column in the case of composite primary keys)

revision number
: an integer, which matches to the revision number in the revision entity table.

revision type
: a small integer

audited fields
: propertied from the original entity being audited

The primary key of the audit table is the combination of the original id of the entity and the revision number, so there can be at most one historic entry for a given entity instance at a given revision.

The current entity data is stored in the original table and in the audit table. This is a duplication of data, however as this solution makes the query system much more powerful, and as memory is cheap, hopefully this won't be a major drawback for the users. A row in the audit table with entity id `ID`, revision `N` and data `D` means: entity with id `ID` has data `D` from revision `N` upwards. Hence, if we want to find an entity at revision `M`, we have to search for a row in the audit table, which has the revision number smaller or equal to `M`, but as large as possible. If no such row is found, or a row with a "deleted" marker is found, it means that the entity didn't exist at that revision.

The "revision type" field can currently have three values: `0`, `1` and `2`, which means `ADD`, `MOD` and `DEL`, respectively. A row with a revision of type `DEL` will only contain the id of the entity and no data (all fields `NULL`), as it only serves as a marker saying "this entity was deleted at that revision".

Additionally, there is a revision entity table which contains the information about the global revision. By default the generated table is named `REVINFO` and contains just two columns: `ID` and `TIMESTAMP`. A row is inserted into this table on each new revision, that is, on each commit of a transaction, which changes audited data. The name of this table can be configured, the name of its columns as well as adding additional columns can be achieved as discussed in Revision Log.

> While global revisions are a good way to provide correct auditing of relations, some people have pointed out that this may be a bottleneck in systems, where data is very often modified. One viable solution is to introduce an option to have an entity "locally revisioned", that is revisions would be created for it independently. This woulld not enable correct versioning of relations, but it would work without the `REVINFO` table. Another possibility is to introduce a notion of "revisioning groups", which would group entities sharing the same revision numbering. Each such group would have to consist of one or more strongly connected components belonging to the entity graph induced by relations between entities. Your opinions on the subject are very welcome on the forum! :)

21.16. Generating schema with Ant

If you would like to generate the database schema file with the Hibernate Tools Ant task, you simply need to use the `org.hibernate.tool.ant.HibernateToolTask` to do so. This task will generate the definitions of all entities, both of which are audited by Envers and those which are not.

For example:

XML

```
<target name="schemaexport" depends="build-demo" description="Exports a generated schema to DB and file">
  <taskdef
    name="hibernatetool"
    classname="org.hibernate.tool.ant.HibernateToolTask"
    classpathref="build.demo.classpath"
  />
  <hibernatetool destdir=".">
    <classpath>
      <fileset refid="lib.hibernate" />
      <path location="${build.demo.dir}" />
      <path location="${build.main.dir}" />
    </classpath>
    <jpaconfiguration persistenceunit="ConsolePU" />
    <hbm2ddl
      drop="false"
      create="true"
      export="false"
      outputfilename="entities-ddl.sql"
      delimiter=";"
      format="true"
    />
  </hibernatetool>
</target>
```

Will generate the following schema:

SQL

```
create table Address (
        id integer generated by default as identity (start with 1),
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        primary key (id)
);

create table Address_AUD (
        id integer not null,
        REV integer not null,
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        REVTYPE tinyint,
        primary key (id, REV)
);

create table Person (
        id integer generated by default as identity (start with 1),
        name varchar(255),
        surname varchar(255),
        address_id integer,
        primary key (id)
);

create table Person_AUD (
        id integer not null,
        REV integer not null,
        name varchar(255),
        surname varchar(255),
        REVTYPE tinyint,
        address_id integer,
        primary key (id, REV)
);

create table REVINFO (
        REV integer generated by default as identity (start with 1),
        REVTSTMP bigint,
        primary key (REV)
);

alter table Person
        add constraint FK8E488775E4C3EA63
        foreign key (address_id)
        references Address;
```

21.17. Mapping exceptions

21.17.1. What isn't and will not be supported

Bags are not supported because they can contain non-unique elements. Persisting, a bag of `String`s violates the relational database principle that each table is a set of tuples.

In case of bags, however (which require a join table), if there is a duplicate element, the two tuples corresponding to the elements will be the same. Hibernate allows this, however Envers (or more precisely: the database connector) will throw an exception when trying to persist two identical elements because of a unique constraint violation.

There are at least two ways out if you need bag semantics:

1. use an indexed collection, with the `@javax.persistence.OrderColumn` annotation
2. provide a unique id for your elements with the `@CollectionId` annotation.

21.17.2. What isn't and *will* be supported

1. Bag style collections with a `@CollectionId` identifier column (see HHH-3950 (https://hibernate.atlassian.net/browse/HHH-3950)).

21.18. `@OneToMany` with `@JoinColumn`

When a collection is mapped using these two annotations, Hibernate doesn't generate a join table. Envers, however, has to do this so that when you read the revisions in which the related entity has changed, you don't get false results.

To be able to name the additional join table, there is a special annotation: `@AuditJoinTable`, which has similar semantics to JPA `@JoinTable`.

One special case are relations mapped with `@OneToMany` with `@JoinColumn` on the one side, and `@ManyToOne` and `@JoinColumn( insertable=false, updatable=false )` on the many side. Such relations are, in fact, bidirectional, but the owning side is the collection.

To properly audit such relations with Envers, you can use the `@AuditMappedBy` annotation. It enables you to specify the reverse property (using the `mappedBy` element). In case of indexed collections, the index column must also be mapped in the referenced entity (using `@Column( insertable=false, updatable=false )`, and specified using `positionMappedBy`. This annotation will affect only the way Envers works. Please note that the annotation is experimental and may change in the future.

21.19. Advanced: Audit table partitioning

21.20. Benefits of audit table partitioning

Because audit tables tend to grow indefinitely, they can quickly become really large. When the audit tables have grown to a certain limit (varying per RDBMS and/or operating system) it makes sense to start using table partitioning. SQL table partitioning offers a lot of advantages including, but certainly not limited to:

1. Improved query performance by selectively moving rows to various partitions (or even purging old rows)
2. Faster data loads, index creation, etc.

21.21. Suitable columns for audit table partitioning

Generally, SQL tables must be partitioned on a column that exists within the table. As a rule it makes sense to use either the *end revision* or the *end revision timestamp* column for partitioning of audit tables.

End revision information is not available for the default `AuditStrategy`.

Therefore the following Envers configuration options are required:

```
org.hibernate.envers.audit_strategy =
org.hibernate.envers.strategy.ValidityAuditStrategy

org.hibernate.envers.audit_strategy_validity_store_revend_timestamp = true
```

Optionally, you can also override the default values using following properties:

```
org.hibernate.envers.audit_strategy_validity_end_rev_field_name

org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name
```

For more information, see Configuration.

The reason why the end revision information should be used for audit table partitioning is based on the assumption that audit tables should be partitioned on an 'increasing level of relevancy', like so:

1. A couple of partitions with audit data that is not very (or no longer) relevant. This can be stored on slow media, and perhaps even be purged eventually.
2. Some partitions for audit data that is potentially relevant.
3. One partition for audit data that is most likely to be relevant. This should be stored on the fastest media, both for reading and writing.

21.22. Audit table partitioning example

In order to determine a suitable column for the 'increasing level of relevancy', consider a simplified example of a salary registration for an unnamed agency.

Currently, the salary table contains the following rows for a certain person X:

Table 11. Salaries table

Year	Salary (USD)
2006	3300

Year	Salary (USD)
2007	3500
2008	4000
2009	4500

The salary for the current fiscal year (2010) is unknown. The agency requires that all changes in registered salaries for a fiscal year are recorded (i.e. an audit trail). The rationale behind this is that decisions made at a certain date are based on the registered salary at that time. And at any time it must be possible reproduce the reason why a certain decision was made at a certain date.

The following audit information is available, sorted on in order of occurrence:

Table 12. Salaries - audit table

Year	Revision type	Revision timestamp	Salary (USD)	End revision timestamp
2006	ADD	2007-04-01	3300	null
2007	ADD	2008-04-01	35	2008-04-02
2007	MOD	2008-04-02	3500	null
2008	ADD	2009-04-01	3700	2009-07-01
2008	MOD	2009-07-01	4100	2010-02-01
2008	MOD	2010-02-01	4000	null
2009	ADD	2010-04-01	4500	null

21.23. Determining a suitable partitioning column

To partition this data, the 'level of relevancy' must be defined. Consider the following:

1. For fiscal year 2006 there is only one revision. It has the oldest *revision timestamp* of all audit rows, but should still be regarded as relevant because it's the latest modification for this fiscal year in the salary table (its *end revision timestamp* is null).

```
  Also, note that it would be very unfortunate if in 2011 there would be an update of the salary for fiscal year 2006
(which is possible in until at least 10 years after the fiscal year),
  and the audit information would have been moved to a slow disk (based on the age of the __revision timestamp__).
  Remember that, in this case, Envers will have to update the _end revision timestamp_ of the most recent audit row.
. There are two revisions in the salary of fiscal year 2007 which both have nearly the same _revision timestamp_ and
a different __end revision timestamp__.
  On first sight, it is evident that the first revision was a mistake and probably not relevant.
  The only relevant revision for 2007 is the one with _end revision timestamp_ null.
```

Based on the above, it is evident that only the *end revision timestamp* is suitable for audit table partitioning. The *revision timestamp* is not suitable.

21.24. Determining a suitable partitioning scheme

A possible partitioning scheme for the salary table would be as follows:

end revision timestamp year = 2008

This partition contains audit data that is not very (or no longer) relevant.

end revision timestamp year = 2009

This partition contains audit data that is potentially relevant.

end revision timestamp year >= 2010 or null

This partition contains the most relevant audit data.

This partitioning scheme also covers the potential problem of the update of the *end revision timestamp*, which occurs if a row in the audited table is modified. Even though Envers will update the *end revision timestamp* of the audit row to the system date at the instant of modification, the audit row will remain in the same partition (the 'extension bucket').

And sometime in 2011, the last partition (or 'extension bucket') is split into two new partitions:

1. *end revision timestamp* year = 2010:: This partition contains audit data that is potentially relevant (in 2011).
2. *end revision timestamp* year >= 2011 or null:: This partition contains the most interesting audit data and is the new 'extension bucket'.

21.25. Envers links

1. Hibernate main page (http://hibernate.org)
2. Forum (http://community.jboss.org/en/envers?view=discussions)
3. JIRA issue tracker (https://hibernate.atlassian.net/) (when adding issues concerning Envers, be sure to select the "envers" component!)
4. IRC channel

5. FAQ (https://community.jboss.org/wiki/EnversFAQ)

22. Database Portability Considerations

22.1. Portability Basics

One of the selling points of Hibernate (and really Object/Relational Mapping as a whole) is the notion of database portability. This could mean an internal IT user migrating from one database vendor to another, or it could mean a framework or deployable application consuming Hibernate to simultaneously target multiple database products by their users. Regardless of the exact scenario, the basic idea is that you want Hibernate to help you run against any number of databases without changes to your code, and ideally without any changes to the mapping metadata.

22.2. Dialect

The first line of portability for Hibernate is the dialect, which is a specialization of the `org.hibernate.dialect.Dialect` contract. A dialect encapsulates all the differences in how Hibernate must communicate with a particular database to accomplish some task like getting a sequence value or structuring a SELECT query. Hibernate bundles a wide range of dialects for many of the most popular databases. If you find that your particular database is not among them, it is not terribly difficult to write your own.

22.3. Dialect resolution

Originally, Hibernate would always require that users specify which dialect to use. In the case of users looking to simultaneously target multiple databases with their build that was problematic. Generally, this required their users to configure the Hibernate dialect or defining their own method of setting that value.

Starting with version 3.2, Hibernate introduced the notion of automatically detecting the dialect to use based on the `java.sql.DatabaseMetaData` obtained from a `java.sql.Connection` to that database. This was much better, expect that this resolution was limited to databases Hibernate know about ahead of time and was in no way configurable or overrideable.

Starting with version 3.3, Hibernate has a fare more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the `org.hibernate.dialect.resolver.DialectResolver` which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```
JAVA

The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies `org.hibernate.exception.JDBCConnectionException` as possibly being thrown. A `JDBCConnectionException` here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

The cool part about these resolvers is that users can also register their own custom resolvers which will be processed ahead of the built-in Hibernate ones. This might be useful in a number of different situations:

- it allows easy integration for auto-detection of dialects beyond those shipped with Hibernate itself
- it allows you to specify to use a custom dialect when a particular database is recognized.

To register one or more resolvers, simply specify them (separated by commas, tabs or spaces) using the 'hibernate.dialect_resolvers' configuration setting (see the `DIALECT_RESOLVERS` constant on `org.hibernate.cfg.Environment`).

22.4. Identifier generation

When considering portability between databases, another important decision is selecting the identifier generation strategy you want to use. Originally Hibernate provided the *native* generator for this purpose, which was intended to select between a *sequence*, *identity*, or *table* strategy depending on the capability of the underlying database.

However, an insidious implication of this approach comes about when targeting some databases which support *identity* generation and some which do not. *identity* generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value. It is what is known as a *post-insert* generation strategy because the insert must actually happen before we can know the identifier value.

Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context, it must then issue the insert immediately when the users requests that the entity be associated with the session (e.g. like via `save()` or `persist()`), regardless of current transactional semantics.

> Hibernate was changed slightly, once the implications of this were better understood, so now the insert could be delayed in cases where this is feasible.
>
> The underlying issue is that the actual semantics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of enhanced (http://in.relation.to/2082.lace) identifier generators targeting portability in a much different way.

There are specifically 2 bundled *enhanced* generators:

- `org.hibernate.id.enhanced.SequenceStyleGenerator`
- `org.hibernate.id.enhanced.TableGenerator`

The idea behind these generators is to port the actual semantics of the identifier value generation to the different databases. For example, the `org.hibernate.id.enhanced.SequenceStyleGenerator` mimics the behavior of a sequence on databases which do not support sequences by using a table.

22.5. Database functions

This is an area in Hibernate in need of improvement. In terms of portability concerns, this function handling currently works pretty well in HQL, however, it is quite lacking in all other aspects.

SQL functions can be referenced in many ways by users. However, not all databases support the same set of functions. Hibernate, provides a means of mapping a *logical* function name to a delegate which knows how to render that particular function, perhaps even using a totally different physical function call.

Technically this function registration is handled through the `org.hibernate.dialect.function.SQLFunctionRegistry` class which is intended to allow users to provide custom function definitions without having to provide a custom dialect. This specific behavior is not fully completed as of yet.

It is sort of implemented such that users can programmatically register functions with the `org.hibernate.cfg.Configuration` and those functions will be recognized for HQL.

22.6. Type mappings

TODO: document the following as well

22.6.1. BLOB/CLOB mappings

22.6.2. Boolean mappings

JPA portability

- HQL/JPQL differences
- naming strategies
- basic types
- simple id types
- generated id types
- composite ids and many-to-one
- "embedded composite identifiers"

23. Configurations

23.1. Strategy configurations

Many configuration settings define pluggable strategies that Hibernate uses for various purposes. The configuration of many of these strategy type settings accept definition in various forms. The documentation of such configuration settings refer here. The types of forms available in such cases include:

short name (if defined)
: Certain built-in strategy implementations have a corresponding short name.

strategy instance
: An instance of the strategy implementation to use can be specified

strategy Class reference
: A `java.lang.Class` reference of the strategy implementation to use

strategy Class name
: The class name (`java.lang.String`) of the strategy implementation to use

23.2. General Configuration

Property	Example	Purpose
`hibernate.dialect`	`org.hibernate.dialect.PostgreSQL94Dialect`	The classname of a Hibernate `Dialect` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/or from which Hibernate can generate SQL optimized for a particular relational databas In most cases Hibernate can choose the correct `Dialect` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/dialect/Dialect.html) implement metadata returned by the JDBC driver.
`hibernate.current_session_context_class`	`jta`, `thread`, `managed`, or a custom class implementing `org.hibernate.context.spi.CurrentSessionContext`	Supply a custom strategy for the scoping of the *current* `Session`. The definition of what exactly *current* means is controlled by the `CurrentSessionCor` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/spi/CurrentSessionContex Note that for backwards compatibility, if a `CurrentSessionContext` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/spi/CurrentSessionContex is configured this will default to the `JTASessionContext` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/context/internal/JTASessionConte

23.3. Database connection properties

Property	Example	Purpose
`hibernate.connection.driver_class` or `javax.persistence.jdbc.driver`	`org.postgresql.Driver`	Names the JDBC `Driver` class name.
`hibernate.connection.url` or `javax.persistence.jdbc.url`	`jdbc:postgresql:hibernate_orm_test`	Names the JDBC connection URL.
`hibernate.connection.username` or `javax.persistence.jdbc.user`		Names the JDBC connection user name.
`hibernate.connection.password` or `javax.persistence.jdbc.password`		Names the JDBC connection password.
`hibernate.connection.isolation`	`REPEATABLE_READ` or `Connection.TRANSACTION_REPEATABLE_READ`	Names the JDBC connection transaction isolation level.
`hibernate.connection.autocommit`	`true` or `false` (default value)	Names the JDBC connection autocommit mode.

`hibernate.connection.datasource`		Either a `javax.sql.DataSource` instance or a JNDI name unde For JNDI names, ses also `hibernate.jndi.class`, `hibernate.`
`hibernate.connection`		Names a prefix used to define arbitrary JDBC connection prope when creating a connection.
`hibernate.connection.provider_class`	`org.hibernate.hikaricp.internal.HikariCPConnectionProvider`	Names the `ConnectionProvider` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/j JDBC connections. Can reference: • an instance of `ConnectionProvider` • a `Class<? extends ConnectionProvider` object reference • a fully qualified name of a class implementing `ConnectionP` The term `class` appears in the setting name due to legacy reas
`hibernate.jndi.class`		Names the JNDI `javax.naming.InitialContext` class.
`hibernate.jndi.url`	java:global/jdbc/default	Names the JNDI provider/connection url.
`hibernate.jndi`		Names a prefix used to define arbitrary JNDI `javax.naming.In` These properties are passed along to `javax.naming.InitialCo`
`hibernate.connection.acquisition_mode`	`immediate`	Specifies how Hibernate should acquire JDBC connections. The `org.hibernate.ConnectionAcquisitionMode`. Should generally only configure this or `hibernate.connection`
`hibernate.connection.release_mode`	`auto` (default value)	Specifies how Hibernate should release JDBC connections. The (`after_transaction` for JDBC transactions and `after_statem` Should generally only configure this or `hibernate.connection`
Hibernate internal connection pool options		
`hibernate.connection.initial_pool_size`	1 (default value)	Minimum number of connections for the built-in Hibernate con
`hibernate.connection.pool_size`	20 (default value)	Maximum number of connections for the built-in Hibernate cor

`hibernate.connection.pool_validation_interval`	30 (default value)	The number of seconds between two consecutive pool validatio based on the connection acquisition request count.

23.4. c3p0 properties

Property	Example	Purpose
`hibernate.c3p0.min_size`	1	Minimum size of C3P0 connection pool. Refers to c3p0 `minPoolSize` setting (http://www.mchange.com/projects/c3p0/#minPoolSize).
`hibernate.c3p0.max_size`	5	Maximum size of C3P0 connection pool. Refers to c3p0 `maxPoolSize` setting (http://www.mchange.com/projects/c3p0/#maxPoolSize).
`hibernate.c3p0.timeout`	30	Maximum idle time for C3P0 connection pool. Refers to c3p0 `maxIdleTime` setting (http://www.mchange.com/projects/c3p0/#maxIdleTime).
`hibernate.c3p0.max_statements`	5	Maximum size of C3P0 statement cache. Refers to c3p0 `maxStatements` setting (http://www.mchange.com/projects/c3p0/#maxStatements).
`hibernate.c3p0.acquire_increment`	2	Number of connections acquired at a time when there's no connection available in the pool. Refers to c3p0 `acquireIncrement` setting (http://www.mchange.com/projects/c3p0/#acquireIncrement).
`hibernate.c3p0.idle_test_period`	5	Idle time before a C3P0 pooled connection is validated. Refers to c3p0 `idleConnectionTestPeriod` setting (http://www.mchange.com/projects/c3p0/#idleConnectionTestPeriod).
`hibernate.c3p0`		A setting prefix used to indicate additional c3p0 properties that need to be passed to the underlying c3p0 connection pool.

23.5. Mapping Properties

Property	Example
Table qualifying options	
`hibernate.default_catalog`	A catalog name

`hibernate.default_schema`	A schema name
`hibernate.schema_name_resolver`	The fully qualified name of an `org.hibernate.engine.jdbc.env.spi.SchemaNameResolver` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/jdbc/env/spi/SchemaNameResolver.
Identifier options	
`hibernate.id.new_generator_mappings`	`true` (default value) or `false`
`hibernate.use_identifier_rollback`	`true` or `false` (default value)
`hibernate.id.optimizer.pooled.preferred`	`none`, `hilo`, `legacy-hilo`, `pooled` (default value), `pooled-lo`, `pooled-lotl` or a fully-qu (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/id/enhanced/Optimizer.html) implementa
Quoting options	
`hibernate.globally_quoted_identifiers`	`true` or `false` (default value)
`hibernate.globally_quoted_identifiers_skip_column_definitions`	`true` or `false` (default value)
`hibernate.auto_quote_keyword`	`true` or `false` (default value)
Discriminator options	
`hibernate.discriminator.implicit_for_joined`	`true` or `false` (default value)

hibernate.discriminator.ignore_explicit_for_joined	true or false (default value)
Naming strategies	
hibernate.implicit_naming_strategy	default (default value), jpa, legacy-jpa, legacy-hbm, component-path
hibernate.physical_naming_strategy	org.hibernate.boot.model.naming. PhysicalNamingStrategyStandardImpl (default valu
Metadata scanning options	

`hibernate.archive.scanner`	
`hibernate.archive.interpreter`	
`hibernate.archive.autodetection`	`hbm,class` (default value)

hibernate.mapping.precedence	hbm,class (default value)
JDBC-related options	
hibernate.use_nationalized_character_data	true or false (default value)
hibernate.jdbc.lob.non_contextual_creation	true or false (default value)
hibernate.jdbc.time_zone	A java.util.TimeZone, a java.time.ZoneId or a String representation of a ZoneId
hibernate.dialect.oracle.prefer_long_raw	true or false (default value)
Bean Validation options	
javax.persistence.validation.factory	javax.validation.ValidationFactory implementation
hibernate.check_nullability	true or false
hibernate.validator.apply_to_ddl	true (default value) or false
Misc options	
hibernate.create_empty_composites.enabled	true or false (default value)
hibernate.entity_dirtiness_strategy	fully-qualified class name or an actual CustomEntityDirtinessStrategy instance
hibernate.default_entity_mode	pojo (default value) or dynamic-map

23.6. Bytecode Enhancement Properties

Property	Example	Purpose
`hibernate.enhancer.enableDirtyTracking`	`true` or `false` (default value)	Enable dirty tracking feature in runtime bytecode enhancement.
`hibernate.enhancer.enableLazyInitialization`	`true` or `false` (default value)	Enable lazy loading feature in runtime bytecode enhancement. This way, even basic types (e.g. `@Ba` `FetchType.LAZY`)) can be fetched lazily.
`hibernate.enhancer.enableAssociationManagement`	`true` or `false` (default value)	Enable association management feature in runtime bytecode enhancement which automatically sy bidirectional association when only one side is changed.
`hibernate.bytecode.provider`	`javassist` (default value)	The `BytecodeProvider` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/bytecode/spi/Byteco built-in implementation flavor. Currently, only `javassist` is supported.
`hibernate.bytecode.use_reflection_optimizer`	`true` or `false` (default value)	Should we use reflection optimization? The reflection optimizer implements the `ReflectionOptir` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/bytecode/spi/ReflectionOptimizer.html) interfa entity instantiation and property getter/setter calls.

23.7. Query settings

Property	Example	Purpose
`hibernate.query.plan_cache_max_size`	`2048` (default value)	The maximum number of entries including: • `HQLQueryPlan` (https://docs.jboss.org/hibernate/orm • `FilterQueryPlan` (https://docs.jboss.org/hibernate/ • `NativeSQLQueryPlan` (https://docs.jboss.org/hiber maintained by `QueryPlanCache` (https://docs.jboss.or
`hibernate.query.plan_parameter_metadata_max_size`	`128` (default value)	The maximum number of strong references associa (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe

hibernate.order_by.default_null_ordering	none, first or last	Defines precedence of null values in ORDER BY cla
hibernate.discriminator.force_in_select	true or false (default value)	For entities which do not explicitly say, should we f
hibernate.query.substitutions	true=1,false=0	A comma-separated list of token substitutions to us
hibernate.query.factory_class	org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory (default value) or org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory	Chooses the HQL parser implementation.
hibernate.query.jpaql_strict_compliance	true or false (default value)	Map from tokens in Hibernate queries to SQL toker Should we strictly adhere to JPA Query Language (J Setting this to true may cause valid HQL to throw a
hibernate.query.startup_check	true (default value) or false	Should named queries be checked during startup?
hibernate.proc.param_null_passing	true or false (default value)	Global setting for whether null parameter bindin (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe being to allow any default argument values to be a This defines a global setting, which can then be con org.hibernate.procedure.ParameterRegistrat Values are true (pass the NULLs) or false (do no
hibernate.jdbc.log.warnings	true or false	Enable fetching JDBC statement warning for loggin org.hibernate.dialect.Dialect#isJdbcLogWar
hibernate.session_factory.statement_inspector	A fully-qualified class name, an instance, or a Class object reference	Names a StatementInspector (https://docs.jboss.org applied to every Session created by the current S Can reference • StatementInspector instance • StatementInspector implementation {@link (• StatementInspector implementation class na
Multi-table bulk HQL operations		

`hibernate.hql.bulk_id_strategy`	A fully-qualified class name, an instance, or a `Class` object reference	Provide a custom `org.hibernate.hql.spi.id.Mu` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe operations.
`hibernate.hql.bulk_id_strategy.global_temporary.drop_tables`	`true` or `false` (default value)	For databases that don't support local tables, but ju bulk HQL operations when the `SessionFactory` o
`hibernate.hql.bulk_id_strategy.persistent.drop_tables`	`true` or `false` (default value)	This configuration property is used by the `Persist` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe which do not support temporary tables. It follows a segment rows from the various sessions. This configuration property allows you to DROP the `EntityManagerFactory` is closed.
`hibernate.hql.bulk_id_strategy.persistent.schema`	Database schema name. By default, the `hibernate.default_schema` is used.	This configuration property is used by the `Persist` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe which do not support temporary tables. It follows a segment rows from the various sessions. This configuration property defines the database sc
`hibernate.hql.bulk_id_strategy.persistent.catalog`	Database catalog name. By default, the `hibernate.default_catalog` is used.	This configuration property is used by the `Persist` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe which do not support temporary tables. It follows a segment rows from the various sessions. This configuration property defines the database ca
`hibernate.legacy_limit_handler`	`true` or `false` (default value)	Setting which indicates whether or not to use `org.h` optimizations to allow legacy 4.x limit behavior. Legacy 4.x behavior favored performing paginatio limit handler behavior favors performance, thus, if
`hibernate.query.conventional_java_constants`	`true` (default value) or `false`	Setting which indicates whether or not Java consta (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/ Default is `true`. Existing applications may want to performance overhead for using non-conventional Check out HHH-4959 (https://hibernate.atlassian.net/br

23.8. Batching properties

Property	Example	Purpose
`hibernate.jdbc.batch_size`	5	Maximum JDB
`hibernate.order_inserts`	`true` or `false` (default value)	Forces Hiberna batching when
`hibernate.order_updates`	`true` or `false` (default value)	Forces Hiberna batching when systems.
`hibernate.jdbc.batch_versioned_data`	`true` (default value) or `false`	Should version Set this proper usually safe, bu data.
`hibernate.batch_fetch_style`	`LEGACY`(default value)	Names the `Bat` (https://docs.jbos Can specify eit (https://docs.jbos `BatchFetchSt` instance. `LEGA`
`hibernate.jdbc.batch.builder`	The fully qualified name of an `BatchBuilder` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/jdbc/batch/spi/BatchBuilder.html) implementation class type or an actual object instance	Names the `Bat` (https://docs.jbos implementatio

23.8.1. Fetching properties

Property	Example	Purpose
`hibernate.max_fetch_depth`	A value between `0` and `3`	Sets a maximum depth for the outer join fetch tree for single-ended associations. A single-ended association is a one-to-one or many-to-one assocation. A value of `0` disables default outer join fetching.
`hibernate.default_batch_fetch_size`	`4` , `8` , or `16`	Default size for Hibernate Batch fetching of associations (lazily fetched associations can be fetched in batches to prevent N+1 query problems).
`hibernate.jdbc.fetch_size`	`0` or an integer	A non-zero value determines the JDBC fetch size, by calling `Statement.setFetchSize()`.

`hibernate.jdbc.use_scrollable_resultset`	`true` or `false`	Enables Hibernate to use JDBC2 scrollable resultsets. This property is only relevant for user-supplied JDBC connections. Otherwise, Hibernate uses connection metadata.
`hibernate.jdbc.use_streams_for_binary`	`true` or `false` (default value)	Use streams when writing or reading `binary` or `serializable` types to or from JDBC. This is a system-level property.
`hibernate.jdbc.use_get_generated_keys`	`true` or `false`	Allows Hibernate to use JDBC3 `PreparedStatement.getGeneratedKeys()` to retrieve natively-generated keys after insert. You need the JDBC3+ driver and JRE1.4+. Disable this property if your driver has problems with the Hibernate identifier generators. By default, it tries to detect the driver capabilities from connection metadata.
`hibernate.jdbc.wrap_result_sets`	`true` or `false` (default value)	Enable wrapping of JDBC result sets in order to speed up column name lookups for broken JDBC drivers.
`hibernate.enable_lazy_load_no_trans`	`true` or `false` (default value)	Initialize Lazy Proxies or Collections outside a given Transactional Persistence Context. Although enabling this configuration can make `LazyInitializationException` go away, it's better to use a fetch plan that guarantees that all properties are properly initialised before the Session is closed. In reality, you shouldn't probably enable this setting anyway.

23.9. Statement logging and statistics

Property	Example	Purpose
SQL statement logging		
`hibernate.show_sql`	`true` or `false` (default value)	Write all SQL statements t console. This is an alterna setting the log category `org.hibernate.SQL` to de

hibernate.format_sql	true or false (default value)	Pretty-print the SQL in the and console.
hibernate.use_sql_comments	true or false (default value)	If true, Hibernate generat comments inside the SQL, easier debugging.
Statistics settings		
hibernate.generate_statistics	true or false	Causes Hibernate to collec statistics for performance
hibernate.stats.factory	The fully qualified name of an StatisticsFactory (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/stat/spi/StatisticsFactory.html) implementation or an actual instance The StatisticsFactory allow you to customize how the Hibernate Statistics are being collected.	hibernate.session.ever

23.10. Cache Properties

Property	Example	Purpose
hibernate.cache.region.factory_class	org.hibernate.cache.infinispan.InfinispanRegionFactory	The fully-qualified name of the RegionFactory implen
hibernate.cache.default_cache_concurrency_strategy		Setting used to give the name of the default CacheConc (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hiberna @javax.persistence.Cacheable or @org.hibernat to override the global setting.
hibernate.cache.use_minimal_puts	true (default value) or false	Optimizes second-level cache operation to minimize w clustered caches and is enabled by default for clustere
hibernate.cache.use_query_cache	true or false (default value)	Enables the query cache. You still need to set individua
hibernate.cache.use_second_level_cache	true (default value) or false	Enable/disable the second level cache, which is enable NoCachingRegionFactory (meaning there is no actua
hibernate.cache.query_cache_factory	Fully-qualified classname	A custom QueryCacheFactory (https://docs.jboss.org/hibe interface. The default is the built-in StandardQueryCa
hibernate.cache.region_prefix	A string	A prefix for second-level cache region names.
hibernate.cache.use_structured_entries	true or false (default value)	Forces Hibernate to store data in the second-level cach

`hibernate.cache.auto_evict_collection_cache`	`true` or `false` (default: false)	Enables the automatic eviction of a bi-directional assoc is added/updated/removed without properly managing
`hibernate.cache.use_reference_entries`	`true` or `false`	Optimizes second-level cache operation to store immut cache directly, this case, lots of disasseble and deep cop
`hibernate.ejb.classcache`	`hibernate.ejb.classcache .org.hibernate.ejb.test.Item = read-write`	Sets the associated entity class cache concurrency strat following pattern `hibernate.ejb.classcache.<fully` strategy used and region the cache region name.
`hibernate.ejb.collectioncache`	`hibernate.ejb.collectioncache .org.hibernate.ejb.test.Item.distributors = read-write, RegionName />`	Sets the associated collection cache concurrency strate following pattern `hibernate.ejb.collectioncache.` the cache strategy used and region the cache region na

23.11. Infinispan properties

Property	Example	Purpose
`hibernate.cache.infinispan.cfg`	`org/hibernate/cache/infinispan/ builder/infinispan-configs.xml`	Classpath or filesystem resource containing the Infinispan configuration settings.
`hibernate.cache.infinispan.statistics`		Property name that controls whether Infinispan statistics are enabled. The property value is expected to be a boolean true or false, and it overrides statistic configuration in base Infinispan configuration, if provided.
`hibernate.cache.infinispan.use_synchronization`		Deprecated setting because Infinispan is designed to always register a `Synchronization` for `TRANSACTIONAL` caches.

`hibernate.cache.infinispan.cachemanager`	There is no default value, the user must specify the property.	Specifies the JNDI name under which the `EmbeddedCacheManager` is bound.

23.12. Transactions properties

Property	Example	Purpose
`hibernate.transaction.jta.platform`	`JBossAS`, `BitronixJtaPlatform`	Names the `JtaPlatform` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engi to use for integrating with JTA systems. Can reference either a `JtaPlatform` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/transaction/jta/platform/sp (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/transaction/jta/platform/sp
`hibernate.jta.prefer_user_transaction`	`true` or `false` (default value)	Should we prefer using the `org.hibernate.engine.transaction.jta.platform.sp` `org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform#retrieveT`
`hibernate.transaction.jta.platform_resolver`		Names the `JtaPlatformResolver` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe implementation to use.
`hibernate.jta.cacheTransactionManager`	`true` (default value) or `false`	A configuration value key used to indicate that it is safe to cache.
`hibernate.jta.cacheUserTransaction`	`true` or `false` (default value)	A configuration value key used to indicate that it is safe to cache.
`hibernate.transaction.flush_before_completion`	`true` or `false` (default value)	Causes the session be flushed during the before completion phase of the transaction. If management instead.
`hibernate.transaction.auto_close_session`	`true` or `false` (default value)	Causes the session to be closed during the after completion phase of the transaction. If management instead.

`hibernate.transaction.coordinator_class`		Names the implementation of `TransactionCoordinatorBuilder` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/resource/transaction/spi/Transacti `TransactionCoordinator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/res
		
		Can be
		
		• `TransactionCoordinatorBuilder` instance
		• `TransactionCoordinatorBuilder` implementation `Class` reference
		• `TransactionCoordinatorBuilder` implementation class name (fully-qualified nar
		
		The following short names are defined for this setting:
		
		`jdbc`
		Manages transactions via calls to `java.sql.Connection` (default for non-JPA applic
		
		`jta`
		Manages transactions via JTA. See Java EE bootstrapping
		
		If a JPA application does not provide a setting for `hibernate.transaction.coordinat` transaction coordinator based on the transaction type for the persistence unit.
		
		If a non-JPA application does not provide a setting for `hibernate.transaction.coord` This default will cause problems if the application actually uses JTA-based transactions should explicitly set `hibernate.transaction.coordinator_class=jta` or provide a (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/resource/transaction/TransactionC `TransactionCoordinator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/res coordinates with JTA-based transactions.

`hibernate.jta.track_by_thread`	`true` (default value) or `false`	A transaction can be rolled back by another thread ("tracking by thread") and not the o transaction timeout handled by a background reaper thread. The ability to handle this situation requires checking the Thread ID every time Session impact.
`hibernate.transaction.factory_class`		This is a legacy setting that's been deprecated and you should use the `hibernate.tran

23.13. Multi-tenancy settings

Property	Example	Purpose

`hibernate.multiTenancy`	`NONE` (default value), `SCHEMA`, `DATABASE`, and `DISCRIMINATOR` (not implemented yet)	The multi-tenancy strategy in use.
`hibernate.multi_tenant_connection_provider`	`true` or `false` (default value)	Names a `MultiTenantConnectionProvider` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/jdbc/connections/spi/Mult `MultiTenantConnectionProvider` is also a service, can be configured directly throu (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/boot/registry/StandardServiceReg
`hibernate.tenant_identifier_resolver`		Names a `CurrentTenantIdentifierResolver` (https://docs.jboss.org/hibernate/orm/5.2/ja implementation to resolve the resolve the current tenant identifier so that calling `Ses` connected to the right tenant. Can be: • `CurrentTenantIdentifierResolver` instance • `CurrentTenantIdentifierResolver` implementation `Class` object reference • `CurrentTenantIdentifierResolver` implementation class name
`hibernate.multi_tenant.datasource.identifier_for_any`	`true` or `false` (default value)	When the `hibernate.connection.datasource` property value is resolved to a `java` the JNDI name used to locate the `DataSource` used for fetching the initial `Connectic` underlying database(s) (in situations where we do not have a tenant id, like startup pr

23.14. Automatic schema generation

Property	Example	Purpose

`hibernate.hbm2ddl.auto`	`none` (default value), `create-only`, `drop`, `create`, `create-drop`, `validate`, and `update`	Setting to perform `SchemaManagementTool` actions automatically as part of `externalHbm2ddlName` value of the `Action` (https://docs.jboss.org/hibernate/o
		
		`none`
		No action will be performed.
		
		`create-only`
		Database creation will be generated.
		
		`drop`
		Database dropping will be generated.
		
		`create`
		Database dropping will be generated followed by database creation.
		
		`create-drop`
		Drop the schema and recreate it on SessionFactory startup. Additionally,
		
		`validate`
		Validate the database schema
		
		`update`
		Update the database schema

`javax.persistence.schema-generation.database.action`	`none` (default value), `create-only`, `drop`, `create`, `create-drop`, `validate`, and `update`	Setting to perform `SchemaManagementTool` actions automatically as part
		`externalJpaName` value of the `Action` (https://docs.jboss.org/hibernate/orm/5
		
		`none`
		No action will be performed.
		
		`create`
		Database creation will be generated.
		
		`drop`
		Database dropping will be generated.
		
		`drop-and-create`
		Database dropping will be generated followed by database creation.

`javax.persistence.schema-generation.scripts.action`	`none` (default value), `create-only`, `drop`, `create`, `create-drop`, `validate`, and `update`	Setting to perform `SchemaManagementTool` actions writing the commands
		value of the `Action` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hiberna
		
		`none`
		No action will be performed.
		
		`create`
		Database creation will be generated.
		
		`drop`
		Database dropping will be generated.
		
		`drop-and-create`
		Database dropping will be generated followed by database creation.

`javax.persistence.schema-generation-connection`		Allows passing a specific `java.sql.Connection` instance to be used by `S`

`javax.persistence.database-product-name`		Specifies the name of the database provider in cases where a Connection t such cases, a value for this setting *must* be specified. The value of this setting is expected to match the value returned by `java.s` Additionally, specifying `javax.persistence.database-major-version` a understand exactly how to generate the required schema commands.
`javax.persistence.database-major-version`		Specifies the major version of the underlying database, as would be return database. This value is used to help more precisely determine how to perform schem `javax.persistence.database-product-name` does not provide enough d
`javax.persistence.database-minor-version`		Specifies the minor version of the underlying database, as would be returr database. This value is used to help more precisely determine how to perform schem `javax.persistence.database-product-name` and `javax.persistence`
`javax.persistence.schema-generation.create-source`		Specifies whether schema generation commands for schema creation are t combination of the two. See `SourceType` (https://docs.jboss.org/hibernate/orm/ If no value is specified, a default is assumed as follows: • if source scripts are specified (per `javax.persistence.schema-gener` • otherwise, `metadata` is assumed
`javax.persistence.schema-generation.drop-source`		Specifies whether schema generation commands for schema dropping are combination of the two. See `SourceType` (https://docs.jboss.org/hibernate/orm/ If no value is specified, a default is assumed as follows: • if source scripts are specified (per `javax.persistence.schema-gener` • otherwise, `metadata` is assumed

javax.persistence.schema-generation.create-script-source		Specifies the create script file as either a java.io.Reader configured fo the DDL script. Hibernate historically also accepted hibernate.hbm2ddl.import_files script-source should be preferred over hibernate.hbm2ddl.import_f
javax.persistence.schema-generation.drop-script-source		Specifies the drop script file as either a java.io.Reader configured for the DDL script.
javax.persistence.schema-generation.scripts.create-target		For cases where the javax.persistence.schema-generation.scripts. DDL script file, javax.persistence.schema-generation.scripts.creat DDL script or a string specifying the file URL for the DDL script.
javax.persistence.schema-generation.scripts.drop-target		For cases where the javax.persistence.schema-generation.scripts. DDL script file, javax.persistence.schema-generation.scripts.drop- script or a string specifying the file URL for the DDL script.
javax.persistence.hibernate.hbm2ddl.import_files	import.sql (default value)	Comma-separated names of the optional files containing SQL DML stateme statements of a give file are executed before the statements of the followin These statements are only executed if the schema is created, meaning that javax.persistence.schema-generation.create-script-source / jav preferred.
javax.persistence.sql-load-script-source		JPA variant of hibernate.hbm2ddl.import_files . Specifies a java.io. file java.net.URL for the SQL load script. A "SQL load script" is a script th
hibernate.hbm2ddl.import_files_sql_extractor		Reference to the ImportSqlCommandExtractor (https://docs.jboss.org/hiberna implementation class to use for parsing source/import files as defined by javax.persistence.schema-generation.drop-script-source or hibe Reference may refer to an instance, a Class implementing ImportSqlComm ImportSqlCommandExtractor implementation. If the fully-qualified nam The default value is SingleLineSqlCommandExtractor (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/hbm2ddl/Single
hibernate.hbm2dll.create_namespaces	true or false (default value)	Specifies whether to automatically create also the database schema/catalog

javax.persistence.create-database-schemas	true or false (default value)	The JPA variant of hibernate.hbm2dll.create_namespaces. Specifies wh to creating database objects (tables, sequences, constraints, etc). The value create schemas in the database or to generate DDL that contains "CREATE S the provider should not attempt to create database schemas.
hibernate.hbm2ddl.schema_filter_provider		Used to specify the SchemaFilterProvider (https://docs.jboss.org/hibernate/o create, drop, migrate, and validate operations on the database sche of these operations to specific namespaces, tables and sequences. All objec
hibernate.hbm2ddl.jdbc_metadata_extraction_strategy	grouped (default value) or individually	Setting to choose the strategy used to access the JDBC Metadata. Valid optio (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/schema/JdbcMe grouped SchemaMigrator (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hiberna (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/schema/spi/S java.sql.DatabaseMetaData#getTables(String, String, String, javax.persistence.Entity have a corresponding mapped database t individually SchemaMigrator (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hiberna (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/schema/spi/S java.sql.DatabaseMetaData#getTables(String, String, String, a corresponding database table exists.
hibernate.hbm2ddl.delimiter	;	Identifies the delimiter to use to separate schema management statements
hibernate.schema_management_tool	A schema name	Used to specify the SchemaManagementTool (https://docs.jboss.org/hibernate/o performing schema management. The default is to use HibernateSchemaM (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/schema/interna
hibernate.synonyms	true or false (default value)	If enabled, allows schema update and validation to support synonyms. Due this is disabled by default.
hibernate.hbm2dll.extra_physical_table_types	BASE TABLE	Identifies a comma-separated list of values to specify extra table types, oth schema update, creation and validation.

`hibernate.schema_update.unique_constraint_strategy`	`DROP_RECREATE_QUIETLY`, `RECREATE_QUIETLY`, `SKIP`	Unique columns and unique keys both use unique constraints in most dial finding existing constraints is extremely inconsistent. Further, non-explicit
		
		Therefore, the `UniqueConstraintSchemaUpdateStrategy` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tool/hbm2ddl/Uniqu
		
		`DROP_RECREATE_QUIETLY`
		Default option. Attempt to drop, then (re-)create each unique constraint.
		
		`RECREATE_QUIETLY`
		Attempts to (re-)create unique constraints, ignoring exceptions thrown i
		
		`SKIP`
		Does not attempt to create unique constraints on a schema update.

`hibernate.hbm2ddl.charset_name`	`Charset.defaultCharset()`	Defines the charset (encoding) used for all input/output schema generation `Charset.defaultCharset()`. This configuration property allows you to ov when reading and writing schema generation resources (e.g. File, URL).
`hibernate.hbm2ddl.halt_on_error`	`true` or `false` (default value)	Whether the schema migration tool should halt on error, therefore termin `SessionFactory` are created even if the schema migration throws except

23.15. Exception handling

Property	Example	Purpose
`hibernate.jdbc.sql_exception_converter`	Fully-qualified name of class implementing `SQLExceptionconverter`	The `SQLExceptionConverter` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/exception/spi/SQLExceptionConverter.htm `SQLExceptions` to Hibernate's `JDBCException` hierarchy. The default is to use the configure (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/dialect/Dialect.html)'s preferred `SQLExc`

23.16. Session events

Property	Example	Purpose
`hibernate.session.events.auto`		Fully qualified class name implementing the `SessionEver` interface.

`hibernate.session_factory.interceptor` or `hibernate.ejb.interceptor`	`org.hibernate.EmptyInterceptor` (default value)	Names a https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe implementation to be applied to every `Session` created b `org.hibernate.SessionFactory` Can reference: • `Interceptor` instance • `Interceptor` implementation `Class` object referenc • `Interceptor` implementation class name
`hibernate.ejb.interceptor.session_scoped`	fully-qualified class name or class reference	An optional Hibernate interceptor. The interceptor instance is specific to a given Session insta not thread-safe) has to implement `org.hibernate.Inter` no-arg constructor. This property can not be combined with `hibernate.ejb.`
`hibernate.ejb.session_factory_observer`	fully-qualified class name or class reference	Specifies a `SessionFactoryObserver` to be applied to the The class must have a no-arg constructor.
`hibernate.ejb.event`	`hibernate.ejb.event.pre-load = com.acme.SecurityListener,com.acme.AuditListener`	Event listener list for a given event type. The list of event l comma separated fully qualified class name list.

23.17. JMX settings

Property	Example	Purpose
`hibernate.jmx.enabled`	`true` or `false` (default value)	Enable JMX.
`hibernate.jmx.usePlatformServer`	`true` or `false` (default value)	Uses the platform MBeanServer as returned by `ManagementFactory#getPlatformMBeanServer()`.
`hibernate.jmx.agentId`		The agent identifier of the associated `MBeanServer`.
`hibernate.jmx.defaultDomain`		The domain name of the associated `MBeanServer`.

`hibernate.jmx.sessionFactoryName`		The `SessionFactory` name appended to the object name the Manageable Bean is registered with. If null, the `hibernate.session_factory_name` configuration value is used.
`org.hibernate.core`		The default object domain appended to the object name the Manageable Bean is registered with.

23.18. JACC settings

Property	Example	Purpose
`hibernate.jacc.enabled`	`true` or `false` (default value)	Is JACC enabled?
`hibernate.jacc`	`hibernate.jacc.allowed.org.jboss.ejb3.test.jacc.AllEntity`	The property name defines the role (e.g. `allowed`) and the entity class name (e.g. `org.jboss.ejb3.test.jacc.AllEntity`), while the property value defines the authorized actions (e.g. `insert,update,read`).
`hibernate.jacc_context_id`		A String identifying the policy context whose PolicyConfiguration interface is to be returned. The value passed to this parameter must not be null.

23.19. ClassLoaders properties

Property	Example	Purpose
`hibernate.classLoaders`		Used to define a `java.util.Collection<ClassLoader>` or the `ClassLoader` instance Hibernate should use for class-loading and resource-lookups.
`hibernate.classLoader.application`		Names the `ClassLoader` used to load user application classes.
`hibernate.classLoader.resources`		Names the `ClassLoader` Hibernate should use to perform resource loading.
`hibernate.classLoader.hibernate`		Names the `ClassLoader` responsible for loading Hibernate classes. By default this is the `ClassLoader` that loaded this class.

`hibernate.classLoader.environment`		Names the `ClassLoader` used when Hibernate is unable to locates classes on the `hibernate.classLoader.application` or `hibernate.classLoader.hibernate`.

23.20. Bootstrap properties

Property	Example
`hibernate.integrator_provider`	The fully qualified name of an `IntegratorProvider` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/boot/spi/IntegratorProvider.html)
`hibernate.strategy_registration_provider`	The fully qualified name of an `StrategyRegistrationProviderList` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/boot/spi/StrategyRegistrationProviderList.html)
`hibernate.type_contributors`	The fully qualified name of an `TypeContributorList` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/boot/spi/TypeContributorList.html)
`hibernate.persister.resolver`	The fully qualified name of a `PersisterClassResolver` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/persister/spi/PersisterClassResolver.html) or a `PersisterClassResc`
`hibernate.persister.factory`	The fully qualified name of a `PersisterFactory` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/persister/spi/Pe a `PersisterFactory` instance
`hibernate.service.allow_crawling`	`true` (default value) or `false`

23.21. Miscellaneous properties

Property	Example	Purpose
`hibernate.dialect_resolvers`		Names any additional `DialectResolver` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibe implementations to register with the standard `DialectFactory` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/engine/jdbc/dialect/spi/DialectFactory.htm
`hibernate.session_factory_name`	A JNDI name	Setting used to name the Hibernate `SessionFactory`. Naming the `SessionFactory` allows the same name is used on each JVM. If `hibernate.session_factory_name_is_jndi` is set to `true`, this is also the name under v startup and from which it can be obtained from JNDI.

`hibernate.session_factory_name_is_jndi`	`true` (default value) or `false`	Does the value defined by `hibernate.session_factory_name` represent a JNDI namespace should be bound and made accessible? Defaults to `true` for backwards compatibility. Set this to `false` if naming a SessionFactory i JNDI context exists in the runtime environment or if the user simply does not want JNDI to b
`hibernate.ejb.entitymanager_factory_name`	By default, the persistence unit name is used, otherwise a randomly generated UUID	Internally, Hibernate keeps track of all `EntityManagerFactory` instances using the `EntityM` to identify a given `EntityManagerFactory` reference.
`hibernate.ejb.cfgfile`	`hibernate.cfg.xml` (default value)	XML configuration file to use to configure Hibernate.
`hibernate.ejb.discard_pc_on_close`	`true` or `false` (default value)	If true, the persistence context will be discarded (think `clear()` when the method is called. C the transaction completion: all objects will remain managed, and any change will be synchro transaction completion).
`hibernate.ejb.metamodel.population`	`enabled` or `disabled`, or `ignoreUnsupported` (default value)	Setting that indicates whether to build the JPA types. Accepts three values: enabled Do the build disabled Do not do the build ignoreUnsupported Do the build, but ignore any non-JPA features that would otherwise result in a failure (e.g.

`hibernate.jpa.static_metamodel.population`	`enabled` or `disabled`, or `skipUnsupported` (default value)	Setting that controls whether we seek out JPA *static metamodel* classes and populate them.
		
		Accepts three values:
		
		`enabled`
		Do the population
		
		`disabled`
		Do not do the population
		
		`skipUnsupported`
		Do the population, but ignore any non-JPA features that would otherwise result in the popu

`hibernate.delay_cdi_access`	`true` or `false` (default value)	Defines delayed access to CDI `BeanManager`. Starting in 5.1 the preferred means for CDI boot (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/jpa/event/spi/jpa/ExtendedBeanManager.

`hibernate.allow_update_outside_transaction`	`true` or `false` (default value)	Setting that allows to perform update operations outside of a transaction boundary.
		
		Accepts two values:
		
		`true`
		allows to flush an update out of a transaction
		
		`false`
		does not allow

`hibernate.collection_join_subquery`	`true` (default value) or `false`	Setting which indicates whether or not the new JOINS over collection tables should be rewrit
`hibernate.allow_refresh_detached_entity`	`true` (default value when using Hibernate native bootstrapping) or `false` (default value when using JPA bootstrapping)	Setting that allows to call `javax.persistence.EntityManager#refresh(entity)` or `Sessi` when the `org.hibernate.Session` is obtained from a JPA `javax.persistence.EntityMan`

`hibernate.event.merge.entity_copy_observer`	`disallow` (default value), `allow`, `log` (testing purpose only) or fully-qualified class name	Setting that specifies how Hibernate will respond when multiple representations of the same merging. The possible values are: **disallow (the default)** throws `IllegalStateException` if an entity copy is detected **allow** performs the merge operation on each entity copy that is detected **log** (provided for testing only) performs the merge operation on each entity copy that is detect setting requires DEBUG logging be enabled for `EntityCopyAllowedLoggedObserver` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/event/internal/EntityCopyAllowedLogg In addition, the application may customize the behavior by providing an implementation of (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/event/spi/EntityCopyObserver.html) and `hibernate.event.merge.entity_copy_observer` to the class name. When this property is entity copy detected while cascading the merge operation. In the process of merging each ent from each entity copy to its associations with `cascade=CascadeType.MERGE` or `CascadeTyp` entity copy will be overwritten when another entity copy is merged. For more details, check out the Merge gotchas section.

23.22. Envers properties

Property	Example	Purpose
`hibernate.envers.autoRegisterListeners`	`true` (default value) or `false`	When set to `false`, the Envers entity listeners are no longer auto-registered, so you need to register them manually during the bootstrap process.
`hibernate.integration.envers.enabled`	`true` (default value) or `false`	Enable or disable the Hibernate Envers `Service` integration.
`hibernate.listeners.envers.autoRegister`		Legacy setting. Use `hibernate.envers.autoRegisterListeners` or `hibernate.integration.envers.enabled` instead.

23.23. Spatial properties

Property	Example	Purpose
`hibernate.integration.spatial.enabled`	`true` (default value) or `false`	Enable or disable the Hibernate Spatial `Service` int
`hibernate.spatial.connection_finder`	`org.geolatte.geom.codec.db.oracle.DefaultConnectionFinder`	Define the fully-qualified name of class implementi `org.geolatte.geom.codec.db.oracle.Connectic` interface.

23.24. Internal properties

The following configuration properties are used internally, and you shouldn't probably have to configured them in your application.

Property	Example	Purpose
`hibernate.enable_specj_proprietary_syntax`	`true` or `false` (default value)	Enable or disable the SpecJ proprietary mapping syntax which differs from JPA specification. Used during performance testing only.
`hibernate.temp.use_jdbc_metadata_defaults`	`true` (default value) or `false`	This setting is used to control whether we should consult the JDBC metadata to determine certain Settings default values when the database may not be available (mainly in tools usage).
`hibernate.connection_provider.injection_data`	`java.util.Map`	Connection provider settings to be injected in the currently configured connection provider.
`hibernate.jandex_index`	`org.jboss.jandex.Index`	Names a Jandex `org.jboss.jandex.Index` instance to use.

24. Mapping annotations

24.1. JPA annotations

24.1.1. @Access

The `@Access` (http://docs.oracle.com/javaee/7/api/javax/persistence/Access.html) annotation is used to specify the access type of the associated entity class, mapped superclass, or embeddable class, or entity attribute.

See the Access type section for more info.

24.1.2. @AssociationOverride

The `@AssociationOverride` (http://docs.oracle.com/javaee/7/api/javax/persistence/AssociationOverride.html) annotation is used to override an association mapping (e.g. `@ManyToOne` , `@OneToOne` , `@OneToMany` , `@ManyToMany`) inherited from a mapped superclass or an embeddable.

24.1.3. @AssociationOverrides

The `@AssociationOverrides` (http://docs.oracle.com/javaee/7/api/javax/persistence/AssociationOverrides.html) is used to group several `@AssociationOverride` annotations.

24.1.4. @AttributeOverride

The `@AttributeOverride` (http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeOverride.html) annotation is used to override an attribute mapping inherited from a mapped superclass or an embeddable.

24.1.5. @AttributeOverrides

The `@AttributeOverrides` (http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeOverrides.html) is used to group several `@AttributeOverride` annotations.

24.1.6. @Basic

The `@Basic` (http://docs.oracle.com/javaee/7/api/javax/persistence/Basic.html) annotation is used to map a basic attribute type to a database column.

See the Basic types chapter for more info.

24.1.7. @Cacheable

The `@Cacheable` (http://docs.oracle.com/javaee/7/api/javax/persistence/Cacheable.html) annotation is used to specify whether an entity should be stored in the second-level cache.

If the `persistence.xml` `shared-cache-mode` XML attribute is set to `ENABLE_SELECTIVE` , then only the entities annotated with the `@Cacheable` are going to be stored in the second-level cache.

If `shared-cache-mode` XML attribute value is `DISABLE_SELECTIVE` , then the entities marked with the `@Cacheable` annotation are not going to be stored in the second-level cache, while all the other entities are stored in the cache.

See the Caching chapter for more info.

24.1.8. @CollectionTable

The `@CollectionTable` (http://docs.oracle.com/javaee/7/api/javax/persistence/CollectionTable.html) annotation is used to specify the database table that stores the values of a basic or an embeddable type collection.

See the Collections of embeddable types section for more info.

24.1.9. `@Column`

The `@Column` (http://docs.oracle.com/javaee/7/api/javax/persistence/Column.html) annotation is used to specify the mapping between a basic entity attribute and the database table column.

See the `@Column` annotation section for more info.

24.1.10. `@ColumnResult`

The `@ColumnResult` (http://docs.oracle.com/javaee/7/api/javax/persistence/ColumnResult.html) annotation is used in conjunction with the `@SqlResultSetMapping` or `@ConstructorResult` annotations to map a SQL column for a given SELECT query.

See the Entity associations with named native queries section for more info.

24.1.11. `@ConstructorResult`

The `@ConstructorResult` (http://docs.oracle.com/javaee/7/api/javax/persistence/ConstructorResult.html) annotation is used in conjunction with the `@SqlResultSetMapping` annotations to map columns of a given SELECT query to a certain object constructor.

24.1.12. `@Convert`

The `@Convert` (http://docs.oracle.com/javaee/7/api/javax/persistence/Convert.html) annotation is used to specify the `AttributeConverter` (http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeConverter.html) implementation used to convert the current annotated basic attribute.

If the `AttributeConverter` uses `autoApply` (http://docs.oracle.com/javaee/7/api/javax/persistence/Converter.html#autoApply--), then all entity attributes with the same target type are going to be converted automatically.

See the `AttributeConverter` section for more info.

24.1.13. `@Converter`

The `@Converter` (http://docs.oracle.com/javaee/7/api/javax/persistence/Converter.html) annotation is used to specify that the current annotate `AttributeConverter` (http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeConverter.html) implementation can be used as a JPA basic attribute converter.

If the `autoApply` (http://docs.oracle.com/javaee/7/api/javax/persistence/Converter.html#autoApply--) attribute is set to `true`, then the JPA provider will automatically convert all basic attributes with the same Java type as defined by the current converter.

See the `AttributeConverter` section for more info.

24.1.14. `@Converts`

The `@Converts` (http://docs.oracle.com/javaee/7/api/javax/persistence/Converts.html) annotation is used to group multiple `@Convert` annotations.

See the `AttributeConverter` section for more info.

24.1.15. `@DiscriminatorColumn`

The `@DiscriminatorColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/DiscriminatorColumn.html) annotation is used to specify the discriminator column name and the discriminator type (http://docs.oracle.com/javaee/7/api/javax/persistence/DiscriminatorColumn.html#discriminatorType--) for the `SINGLE_TABLE` and `JOINED` Inheritance strategies.

See the Discriminator section for more info.

24.1.16. `@DiscriminatorValue`

The `@DiscriminatorValue` (http://docs.oracle.com/javaee/7/api/javax/persistence/DiscriminatorValue.html) annotation is used to specify what value of the discriminator column is used for mapping the current annotated entity.

See the Discriminator section for more info.

24.1.17. `@ElementCollection`

The `@ElementCollection` (http://docs.oracle.com/javaee/7/api/javax/persistence/ElementCollection.html) annotation is used to specify a collection of a basic or embeddable types.

See the Collections section for more info.

24.1.18. `@Embeddable`

The `@Embeddable` (http://docs.oracle.com/javaee/7/api/javax/persistence/Embeddable.html) annotation is used to specify embeddable types. Like basic types, embeddable types do not have any identity, being managed by their owning entity.

See the Embeddables section for more info.

24.1.19. `@Embedded`

The `@Embedded` (http://docs.oracle.com/javaee/7/api/javax/persistence/Embedded.html) annotation is used to specify that a given entity attribute represents an embeddable type.

See the Embeddables section for more info.

24.1.20. `@EmbeddedId`

The `@EmbeddedId` (http://docs.oracle.com/javaee/7/api/javax/persistence/EmbeddedId.html) annotation is used to specify the entity identifier is an embeddable type.

See the Composite identifiers with `@EmbeddedId` section for more info.

24.1.21. `@Entity`

The `@Entity` (http://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html) annotation is used to specify that the currently annotate class represents an entity type. Unlike basic and embeddable types, entity types have an identity and their state is managed by the underlying Persistence Context.

See the Entity section for more info.

24.1.22. `@EntityListeners`

The `@EntityListeners` (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityListeners.html) annotation is used to specify an array of callback listener classes that are used by the current annotated entity.

See the JPA callbacks section for more info.

24.1.23. `@EntityResult`

The `@EntityResult` (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityResult.html) annotation is used with the `@SqlResultSetMapping` annotation to map the selected columns to an entity.

See the Entity associations with named native queries section for more info.

24.1.24. `@Enumerated`

The `@Enumerated` (http://docs.oracle.com/javaee/7/api/javax/persistence/Enumerated.html) annotation is used to specify that an entity attribute represents an enumerated type.

See the `@Enumerated` basic type section for more info.

24.1.25. `@ExcludeDefaultListeners`

The `@ExcludeDefaultListeners` (http://docs.oracle.com/javaee/7/api/javax/persistence/ExcludeDefaultListeners.html) annotation is used to specify that the current annotated entity skips the invocation of any default listener.

24.1.26. `@ExcludeSuperclassListeners`

The `@ExcludeSuperclassListeners` (http://docs.oracle.com/javaee/7/api/javax/persistence/ExcludeSuperclassListeners.html) annotation is used to specify that the current annotated entity skips the invocation of listeners declared by its superclass.

24.1.27. `@FieldResult`

The `@FieldResult` (http://docs.oracle.com/javaee/7/api/javax/persistence/FieldResult.html) annotation is used with the `@EntityResult` annotation to map the selected columns to the fields of some specific entity.

See the Entity associations with named native queries section for more info.

24.1.28. `@ForeignKey`

The `@ForeignKey` (http://docs.oracle.com/javaee/7/api/javax/persistence/ForeignKey.html) annotation is used to specify the associated foreign key of a `@JoinColumn` mapping. The `@ForeignKey` annotation is only used if the automated schema generation tool is enabled, in which case, it allows you to customize the underlying foreign key definition.

See the `@ManyToOne` with `@ForeignKey` section for more info.

24.1.29. `@GeneratedValue`

The `@GeneratedValue` (http://docs.oracle.com/javaee/7/api/javax/persistence/GeneratedValue.html) annotation specifies that the entity identifier value is automatically generated using an identity column, a database sequence, or a table generator. Hibernate supports the `@GeneratedValue` mapping even for `UUID` identifiers.

See the Automatically-generated identifiers section for more info.

24.1.30. `@Id`

The `@Id` (http://docs.oracle.com/javaee/7/api/javax/persistence/Id.html) annotation specifies the entity identifier. An entity must always have an identifier attribute which is used when loading the entity in a given Persistence Context.

See the Identifiers section for more info.

24.1.31. `@IdClass`

The `@IdClass` (http://docs.oracle.com/javaee/7/api/javax/persistence/IdClass.html) annotation is used if the current entity defined a composite identifier. A separate class encapsulates all the identifier attributes, which are mirrored by the current entity mapping.

See the Composite identifiers with `@IdClass` section for more info.

24.1.32. `@Index`

The `@Index` (http://docs.oracle.com/javaee/7/api/javax/persistence/Index.html) annotation is used by the automated schema generation tool to create a database index.

24.1.33. `@Inheritance`

The `@Inheritance` (http://docs.oracle.com/javaee/7/api/javax/persistence/Inheritance.html) annotation is used to specify the inheritance strategy of a given entity class hierarchy.

See the Inheritance section for more info.

24.1.34. `@JoinColumn`

The `@JoinColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/JoinColumn.html) annotation is used to specify the FOREIGN KEY column used when joining an entity association or an embeddable collection.

See the `@ManyToOne` with `@JoinColumn` section for more info.

24.1.35. `@JoinColumns`

The `@JoinColumns` (http://docs.oracle.com/javaee/7/api/javax/persistence/JoinColumns.html) annotation is used to group multiple `@JoinColumn` annotations, which are used when mapping entity association or an embeddable collection using a composite identifier

24.1.36. `@JoinTable`

The `@JoinTable` (http://docs.oracle.com/javaee/7/api/javax/persistence/JoinTable.html) annotation is used to specify the link table between two other database tables.

See the `@JoinTable` mapping section for more info.

24.1.37. `@Lob`

The `@Lob` (http://docs.oracle.com/javaee/7/api/javax/persistence/Lob.html) annotation is used to specify that the current annotated entity attribute represents a large object type.

See the `BLOB` mapping section for more info.

24.1.38. `@ManyToMany`

The `@ManyToMany` (http://docs.oracle.com/javaee/7/api/javax/persistence/ManyToMany.html) annotation is used to specify a many-to-many database relationship.

See the `@ManyToMany` mapping section for more info.

24.1.39. `@ManyToOne`

The `@ManyToOne` (http://docs.oracle.com/javaee/7/api/javax/persistence/ManyToOne.html) annotation is used to specify a many-to-one database relationship.

See the `@ManyToOne` mapping section for more info.

24.1.40. `@MapKey`

The `@MapKey` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKey.html) annotation is used to specify the key of a `java.util.Map` association for which the key type is either the primary key or an attribute of the entity which represents the value of the map.

See the `@MapKey` mapping section for more info.

24.1.41. `@MapKeyClass`

The `@MapKeyClass` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyClass.html) annotation is used to specify the type of the map key of a `java.util.Map` associations.

24.1.42. `@MapKeyColumn`

The `@MapKeyColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyColumn.html) annotation is used to specify the database column which stores the key of a `java.util.Map` association for which the map key is a basic type.

24.1.43. @MapKeyEnumerated

The `@MapKeyEnumerated` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyEnumerated.html) annotation is used to specify that the key of `java.util.Map` association is a Java Enum.

See the `@MapKeyEnumerated` mapping section for more info.

24.1.44. @MapKeyJoinColumn

The `@MapKeyJoinColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyJoinColumn.html) annotation is used to specify that the key of `java.util.Map` association is an entity association. The map key column is a FOREIGN KEY in a link table that also joins the `Map` owner's table with the table where the `Map` value resides.

See the `@MapKeyJoinColumn` mapping section for more info.

24.1.45. @MapKeyJoinColumns

The `@MapKeyJoinColumns` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyJoinColumns.html) annotation is used to group several `@MapKeyJoinColumn` mappings when the `java.util.Map` association key uses a composite identifier.

24.1.46. @MapKeyTemporal

The `@MapKeyTemporal` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapKeyTemporal.html) annotation is used to specify that the key of `java.util.Map` association is a `@TemporalType` (http://docs.oracle.com/javaee/7/api/javax/persistence/TemporalType.html) (e.g. `DATE`, `TIME`, `TIMESTAMP`).

See the `@MapKeyTemporal` mapping section for more info.

24.1.47. @MappedSuperclass

The `@MappedSuperclass` (http://docs.oracle.com/javaee/7/api/javax/persistence/MappedSuperclass.html) annotation is used to specify that the current annotated type attributes are inherited by any subclass entity.

See the `@MappedSuperclass` section for more info.

24.1.48. @MapsId

The `@MapsId` (http://docs.oracle.com/javaee/7/api/javax/persistence/MapsId.html) annotation is used to specify that the entity identifier is mapped by the current annotated `@ManyToOne` or `@OneToOne` association.

See the `@MapsId` mapping section for more info.

24.1.49. @NamedAttributeNode

The `@NamedAttributeNode` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedAttributeNode.html) annotation is used to specify each individual attribute node that needs to be fetched by an Entity Graph.

See the Fetch graph section for more info.

24.1.50. @NamedEntityGraph

The `@NamedEntityGraph` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedEntityGraph.html) annotation is used to specify an Entity Graph that can be used by an entity query to override the default fetch plan.

See the Fetch graph section for more info.

24.1.51. @NamedEntityGraphs

The `@NamedEntityGraphs` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedEntityGraphs.html) annotation is used to group multiple `@NamedEntityGraph` annotations.

24.1.52. @NamedNativeQueries

The `@NamedNativeQueries` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedNativeQueries.html) annotation is used to group multiple `@NamedNativeQuery` annotations.

See the Custom CRUD mapping section for more info.

24.1.53. @NamedNativeQuery

The `@NamedNativeQuery` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedNativeQuery.html) annotation is used to specify a native SQL query that can be retrieved later by its name.

See the Custom CRUD mapping section for more info.

24.1.54. @NamedQueries

The `@NamedQueries` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedQueries.html) annotation is used to group multiple `@NamedQuery` annotations.

24.1.55. @NamedQuery

The `@NamedQuery` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedQuery.html) annotation is used to specify a JPQL query that can be retrieved later by its name.

24.1.56. @NamedStoredProcedureQueries

The `@NamedStoredProcedureQueries` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedStoredProcedureQueries.html) annotation is used to group multiple `@NamedStoredProcedureQuery` annotations.

24.1.57. @NamedStoredProcedureQuery

The `@NamedStoredProcedureQuery` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedStoredProcedureQuery.html) annotation is used to specify a stored procedure query that can be retrieved later by its name.

24.1.58. @NamedSubgraph

The `@NamedSubgraph` (http://docs.oracle.com/javaee/7/api/javax/persistence/NamedSubgraph.html) annotation used to specify a subgraph in an Entity Graph.

24.1.59. `@OneToMany`

The `@OneToMany` (http://docs.oracle.com/javaee/7/api/javax/persistence/OneToMany.html) annotation is used to specify a one-to-many database relationship.

See the `@OneToMany` mapping section for more info.

24.1.60. `@OneToOne`

The `@OneToOne` (http://docs.oracle.com/javaee/7/api/javax/persistence/OneToOne.html) annotation is used to specify a one-to-one database relationship.

See the `@OneToOne` mapping section for more info.

24.1.61. `@OrderBy`

The `@OrderBy` (http://docs.oracle.com/javaee/7/api/javax/persistence/OrderBy.html) annotation is used to specify the entity attributes used for sorting when fetching the current annotated collection.

See the `@OrderBy` mapping section for more info.

24.1.62. `@OrderColumn`

The `@OrderColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/OrderColumn.html) annotation is used to specify that the current annotation collection order should be materialized in the database.

See the `@OrderColumn` mapping section for more info.

24.1.63. `@PersistenceContext`

The `@PersistenceContext` (http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceContext.html) annotation is used to specify the `EntityManager` that needs to be injected as a dependency.

24.1.64. `@PersistenceContexts`

The `@PersistenceContexts` (http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceContexts.html) annotation is used to group multiple `@PersistenceContext` annotations.

24.1.65. `@PersistenceProperty`

The `@PersistenceProperty` (http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceProperty.html) annotation is used by the `@PersistenceContext` annotation to declare JPA provider properties that are passed to the underlying container when the `EntityManager` instance is created.

24.1.66. `@PersistenceUnit`

The `@PersistenceUnit` (http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceUnit.html) annotation is used to specify the `EntityManagerFactory` that needs to be injected as a dependency.

24.1.67. `@PersistenceUnits`

The `@PersistenceUnits` (http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceUnits.html) annotation is used to group multiple `@PersistenceUnit` annotations.

24.1.68. `@PostLoad`

The `@PostLoad` (http://docs.oracle.com/javaee/7/api/javax/persistence/PostLoad.html) annotation is used to specify a callback method that fires after an entity is loaded.

See the JPA callbacks section for more info.

24.1.69. `@PostPersist`

The `@PostPersist` (http://docs.oracle.com/javaee/7/api/javax/persistence/PostPersist.html) annotation is used to specify a callback method that fires after an entity is persisted.

See the JPA callbacks section for more info.

24.1.70. `@PostRemove`

The `@PostRemove` (http://docs.oracle.com/javaee/7/api/javax/persistence/PostRemove.html) annotation is used to specify a callback method that fires after an entity is removed.

See the JPA callbacks section for more info.

24.1.71. `@PostUpdate`

The `@PostUpdate` (http://docs.oracle.com/javaee/7/api/javax/persistence/PostUpdate.html) annotation is used to specify a callback method that fires after an entity is updated.

See the JPA callbacks section for more info.

24.1.72. `@PrePersist`

The `@PrePersist` (http://docs.oracle.com/javaee/7/api/javax/persistence/PrePersist.html) annotation is used to specify a callback method that fires before an entity is persisted.

See the JPA callbacks section for more info.

24.1.73. `@PreRemove`

The `@PreRemove` (http://docs.oracle.com/javaee/7/api/javax/persistence/PreRemove.html) annotation is used to specify a callback method that fires before an entity is removed.

See the JPA callbacks section for more info.

24.1.74. `@PreUpdate`

The `@PreUpdate` (http://docs.oracle.com/javaee/7/api/javax/persistence/PreUpdate.html) annotation is used to specify a callback method that fires before an entity is updated.

See the JPA callbacks section for more info.

24.1.75. `@PrimaryKeyJoinColumn`

The `@PrimaryKeyJoinColumn` (http://docs.oracle.com/javaee/7/api/javax/persistence/PrimaryKeyJoinColumn.html) annotation is used to specify that the primary key column of the current annotated entity is also a foreign key to some other entity (e.g. a base class table in a `JOINED` inheritance strategy, the primary table in a secondary table mapping, or the parent table in a `@OneToOne` relationship).

See the `@PrimaryKeyJoinColumn` mapping section for more info.

24.1.76. `@PrimaryKeyJoinColumns`

The `@PrimaryKeyJoinColumns` (http://docs.oracle.com/javaee/7/api/javax/persistence/PrimaryKeyJoinColumns.html) annotation is used to group multiple `@PrimaryKeyJoinColumn` annotations.

24.1.77. `@QueryHint`

The `@QueryHint` (http://docs.oracle.com/javaee/7/api/javax/persistence/QueryHint.html) annotation is used to specify a JPA provider hint used by a `@NamedQuery` or a `@NamedNativeQuery` annotation.

24.1.78. `@SecondaryTable`

The `@SecondaryTable` (http://docs.oracle.com/javaee/7/api/javax/persistence/SecondaryTable.html) annotation is used to specify a secondary table for the current annotated entity.

See the `@SecondaryTable` mapping section for more info.

24.1.79. `@SecondaryTables`

The `@SecondaryTables` (http://docs.oracle.com/javaee/7/api/javax/persistence/SecondaryTables.html) annotation is used to group multiple `@SecondaryTable` annotations.

24.1.80. `@SequenceGenerator`

The `@SequenceGenerator` (http://docs.oracle.com/javaee/7/api/javax/persistence/SequenceGenerator.html) annotation is used to specify the database sequence used by the identifier generator of the current annotated entity.

24.1.81. `@SqlResultSetMapping`

The `@SqlResultSetMapping` (http://docs.oracle.com/javaee/7/api/javax/persistence/SqlResultSetMapping.html) annotation is used to specify the `ResultSet` mapping of a native SQL query or stored procedure.

See the `SqlResultSetMapping` mapping section for more info.

24.1.82. `@SqlResultSetMappings`

The `@SqlResultSetMappings` (http://docs.oracle.com/javaee/7/api/javax/persistence/SqlResultSetMappings.html) annotation is group multiple `@SqlResultSetMapping` annotations.

24.1.83. `@StoredProcedureParameter`

The `@StoredProcedureParameter` (http://docs.oracle.com/javaee/7/api/javax/persistence/StoredProcedureParameter.html) annotation is used to specify a parameter of a `@NamedStoredProcedureQuery` .

24.1.84. `@Table`

The `@Table` (http://docs.oracle.com/javaee/7/api/javax/persistence/Table.html) annotation is used to specify the primary table of the current annotated entity.

See the `@Table` mapping section for more info.

24.1.85. `@TableGenerator`

The `@TableGenerator` (http://docs.oracle.com/javaee/7/api/javax/persistence/TableGenerator.html) annotation is used to specify the database table used by the identity generator of the current annotated entity.

24.1.86. `@Temporal`

The `@Temporal` (http://docs.oracle.com/javaee/7/api/javax/persistence/Temporal.html) annotation is used to specify the `TemporalType` of the current annotated `java.util.Date` or `java.util.Calendar` entity attribute.

See the Basic temporal types chapter for more info.

24.1.87. `@Transient`

The `@Transient` (http://docs.oracle.com/javaee/7/api/javax/persistence/Transient.html) annotation is used to specify that a given entity attribute should not be persisted.

See the `@Transient` mapping section for more info.

24.1.88. `@UniqueConstraint`

The `@UniqueConstraint` (http://docs.oracle.com/javaee/7/api/javax/persistence/UniqueConstraint.html) annotation is used to specify a unique constraint to be included by the automated schema generator for the primary or secondary table associated with the current annotated entity.

24.1.89. `@Version`

The `@Version` (http://docs.oracle.com/javaee/7/api/javax/persistence/Version.html) annotation is used to specify the version attribute used for optimistic locking.

See the Optimistic locking mapping section for more info.

24.2. Hibernate annotations

24.2.1. ~~@AccessType~~

The ~~@AccessType~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/AccessType.html) annotation is deprecated. You should use either the JPA `@Access` or the Hibernate native `@AttributeAccessor` annotation.

24.2.2. @Any

The `@Any` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Any.html) annotation is used to define the **any-to-one** association which can point to one one of several entity types.

See the `@Any` mapping section for more info.

24.2.3. @AnyMetaDef

The `@AnyMetaDef` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/AnyMetaDef.html) annotation is used to provide metadata about an `@Any` or `@ManyToAny` mapping.

See the `@Any` mapping section for more info.

24.2.4. @AnyMetaDefs

The `@AnyMetaDefs` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/AnyMetaDefs.html) annotation is used to group multiple `@AnyMetaDef` annotations.

24.2.5. @AttributeAccessor

The `@AttributeAccessor` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/AttributeAccessor.html) annotation is used to specify a custom `PropertyAccessStrategy` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/property/access/spi/PropertyAccessStrategy.html).

Should only be used to name a custom `PropertyAccessStrategy`. For property/field access type, the JPA `@Access` annotation should be preferred.

However, if this annotation is used with either value="property" or value="field", it will act just as the corresponding usage of the JPA `@Access` annotation.

24.2.6. @BatchSize

The `@BatchSize` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/BatchSize.html) annotation is used to specify the size for batch loading the entries of a lazy collection.

See the Batch fetching section for more info.

24.2.7. @Cache

The `@Cache` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Cache.html) annotation is used to specify the `CacheConcurrencyStrategy` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/CacheConcurrencyStrategy.html) of a root entity or a collection.

See the Caching chapter for more info.

24.2.8. @Cascade

The `@Cascade` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Cascade.html) annotation is used to apply the Hibernate specific `CascadeType` (http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/CascadeType.html) strategies (e.g. `CascadeType.LOCK`, `CascadeType.SAVE_UPDATE`, `CascadeType.REPLICATE`) on a given association.

For JPA cascading, prefer using the `javax.persistence.CascadeType` (http://docs.oracle.com/javaee/7/api/javax/persistence/CascadeType.html) instead.

When combining both JPA and Hibernate `CascadeType` strategies, Hibernate will merge both sets of cascades.

See the Cascading chapter for more info.

24.2.9. @Check

The `@Check` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Check.html) annotation is used to specify an arbitrary SQL CHECK constraint which can be defined at the class level.

See the Database-level checks chapter for more info.

24.2.10. @CollectionId

The `@CollectionId` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/CollectionId.html) annotation is used to specify an identifier column for an idbag collection.

You might want to use the JPA `@OrderColumn` instead.

24.2.11. @CollectionType

The `@CollectionType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/CollectionType.html) annotation is used to specify a custom collection type.

The collection can also name a `@Type`, which defines the Hibernate Type of the collection elements.

See the Custom collection types chapter for more info.

24.2.12. @ColumnDefault

The `@ColumnDefault` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ColumnDefault.html) annotation is used to specify the `DEFAULT` DDL value to apply when using the automated schema generator.

The same behavior can be achieved using the `definition` attribute of the JPA `@Column` annotation.

See the Default value for database column chapter for more info.

24.2.13. @Columns

The `@Columns` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Columns.html) annotation is used to group multiple JPA `@Column` annotations.

See the `@Columns` mapping section for more info.

24.2.14. `@ColumnTransformer`

The `@ColumnTransformer` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ColumnTransformer.html) annotation is used to customize how a given column value is read from or write into the database.

See the `@ColumnTransformer` mapping section for more info.

24.2.15. `@ColumnTransformers`

The `@ColumnTransformers` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ColumnTransformers.html) annotation iis used to group multiple `@ColumnTransformer` annotations.

24.2.16. `@CreationTimestamp`

The `@CreationTimestamp` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/CreationTimestamp.html) annotation is used to specify that the current annotated temporal type must be initialized with the current JVM timestamp value.

See the `@CreationTimestamp` mapping section for more info.

24.2.17. `@DiscriminatorFormula`

The `@DiscriminatorFormula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/DiscriminatorFormula.html) annotation is used to specify a Hibernate `@Formula` to resolve the inheritance discriminator value.

See the `@DiscriminatorFormula` section for more info.

24.2.18. `@DiscriminatorOptions`

The `@DiscriminatorOptions` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/DiscriminatorOptions.html) annotation is used to provide the `force` and `insert` Discriminator properties.

See the Discriminator section for more info.

24.2.19. `@DynamicInsert`

The `@DynamicInsert` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/DynamicInsert.html) annotation is used to specify that the `INSERT` SQL statement should be generated whenever an entity is to be persisted.

By default, Hibernate uses a cached `INSERT` statement that sets all table columns. When the entity is annotated with the `@DynamicInsert` annotation, the `PreparedStatement` is going to include only the non-null columns.

See the `@CreationTimestamp` mapping section for more info on how `@DynamicInsert` works.

24.2.20. `@DynamicUpdate`

The `@DynamicUpdate` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/DynamicUpdate.html) annotation is used to specify that the `UPDATE` SQL statement should be generated whenever an entity is modified.

By default, Hibernate uses a cached `UPDATE` statement that sets all table columns. When the entity is annotated with the `@DynamicUpdate` annotation, the `PreparedStatement` is going to include only the columns whose values have been changed.

See the `OptimisticLockType.DIRTY` mapping section for more info on how `@DynamicUpdate` works.

> For reattachment of detached entities, the dynamic update is not possible without having the `@SelectBeforeUpdate` annotation as well.

24.2.21. ~~`@Entity`~~

The ~~`@Entity`~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Entity.html) annotation is deprecated. Use the JPA `@Entity` annotation instead.

24.2.22. `@Fetch`

The `@Fetch` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Fetch.html) annotation is used to specify the Hibernate specific `FetchMode` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FetchMode.html) (e.g. `JOIN`, `SELECT`, `SUBSELECT`) used for the current annotated association:

See the `@Fetch` mapping section for more info.

24.2.23. `@FetchProfile`

The `@FetchProfile` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FetchProfile.html) annotation is used to specify a custom fetching profile, similar to a JPA Entity Graph.

See the Fetch mapping section for more info.

24.2.24. `@FetchProfile.FetchOverride`

The `@FetchProfile.FetchOverride` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FetchProfile.FetchOverride.html) annotation is used in conjunction with the `@FetchProfile` annotation, and it's used for overriding the fetching strategy of a particular entity association.

See the Fetch profile section for more info.

24.2.25. `@FetchProfiles`

The `@FetchProfiles` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FetchProfiles.html) annotation is used to group multiple `@FetchProfile` annotations.

24.2.26. `@Filter`

The `@Filter` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Filter.html) annotation is used to add filters to an entity or the target entity of a collection.

See the Filter mapping section for more info.

24.2.27. `@FilterDef`

The `@FilterDef` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FilterDef.html) annotation is used to specify a `@Filter` definition (name, default condition and parameter types, if any).

See the Filter mapping section for more info.

24.2.28. `@FilterDefs`

The `@FilterDefs` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FilterDefs.html) annotation is used to group multiple `@FilterDef` annotations.

24.2.29. `@FilterJoinTable`

The `@FilterJoinTable` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FilterJoinTable.html) annotation is used to add `@Filter` capabilities to a join table collection.

See the FilterJoinTable mapping section for more info.

24.2.30. `@FilterJoinTables`

The `@FilterJoinTables` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/FilterJoinTables.html) annotation is used to group multiple `@FilterJoinTable` annotations.

24.2.31. `@Filters`

The `@Filters` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Filters.html) annotation is used to group multiple `@Filter` annotations.

24.2.32. ~~`@ForeignKey`~~

The ~~`@ForeignKey`~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ForeignKey.html) annotation is deprecated. Use the JPA 2.1 `@ForeignKey` annotation instead.

24.2.33. `@Formula`

The `@Formula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Formula.html) annotation is used to specify an SQL fragment that is executed in order to populate a given entity attribute.

See the `@Formula` mapping section for more info.

24.2.34. `@Generated`

The `@Generated` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Generated.html) annotation is used to specify that the current annotated entity attribute is generated by the database.

See the `@Generated` mapping section for more info.

24.2.35. `@GeneratorType`

The `@GeneratorType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GeneratorType.html) annotation is used to provide a `ValueGenerator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tuple/ValueGenerator.html) and a `GenerationTime` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenerationTime.html) for the current annotated generated attribute.

See the `@GeneratorType` mapping section for more info.

24.2.36. `@GenericGenerator`

The `@GenericGenerator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenericGenerator.html) annotation can be used to configure any Hibernate identifier generator.

See the `@GenericGenerator` mapping section for more info.

24.2.37. `@GenericGenerators`

The `@GenericGenerators` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/GenericGenerators.html) annotation is used to group multiple `@GenericGenerator` annotations.

24.2.38. `@Immutable`

The `@Immutable` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Immutable.html) annotation is used to specify that the annotated entity, attribute, or collection is immutable.

See the `@Immutable` mapping section for more info.

24.2.39. ~~`@Index`~~

The ~~`@Index`~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Index.html) annotation is deprecated. Use the JPA `@Index` annotation instead.

24.2.40. ~~`@IndexColumn`~~

The ~~`@IndexColumn`~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/IndexColumn.html) annotation is deprecated. Use the JPA `@OrderColumn` annotation instead.

24.2.41. @JoinColumnOrFormula

The `@JoinColumnOrFormula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/JoinColumnOrFormula.html) annotation is used to specify that the entity association is resolved either through a FOREIGN KEY join (e.g. `@JoinColumn`) or using the result of a given SQL formula (e.g. `@JoinFormula`).

See the `@JoinColumnOrFormula` mapping section for more info.

24.2.42. @JoinColumnsOrFormulas

The `@JoinColumnsOrFormulas` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/JoinColumnsOrFormulas.html) annotation is used to group multiple `@JoinColumnOrFormula` annotations.

24.2.43. @JoinFormula

The `@JoinFormula` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/JoinFormula.html) annotation is used as a replacement for `@JoinColumn` when the association does not have a dedicated FOREIGN KEY column.

See the `@JoinFormula` mapping section for more info.

24.2.44. @LazyCollection

The `@LazyCollection` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyCollection.html) annotation is used to specify the lazy fetching behavior of a given collection. The possible values are given by the `LazyCollectionOption` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyCollectionOption.html) enumeration:

`TRUE`
: Load it when the state is requested.

`FALSE`
: Eagerly load it.

`EXTRA`
: Prefer extra queries over full collection loading.

The `TRUE` and `FALSE` values are deprecated since you should be using the JPA `FetchType` (http://docs.oracle.com/javaee/7/api/javax/persistence/FetchType.html) attribute of the `@ElementCollection` , `@OneToMany` , or `@ManyToMany` collection.

The `EXTRA` value has no equivalent in the JPA specification, and it's used to avoid loading the entire collection even when the collection is accessed for the first time. Each element is fetched individually using a secondary query.

See the `@LazyCollection` mapping section for more info.

24.2.45. @LazyGroup

The `@LazyGroup` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyGroup.html) annotation is used to specify that an entity attribute should be fetched along with all the other attributes belonging to the same group.

To load entity attributes lazily, bytecode enhancement is needed. By default, all non-collection attributes are loaded in one group named "DEFAULT".

This annotation allows defining different groups of attributes to be initialized together when access one attribute in the group.

See the `@LazyGroup` mapping section for more info.

24.2.46. `@LazyToOne`

The `@LazyToOne` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyToOne.html) annotation is used to specify the laziness options, represented by `LazyToOneOption` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/LazyToOneOption.html), available for a `@OneToOne` or `@ManyToOne` association.

`LazyToOneOption` defines the following alternatives:

FALSE
: Eagerly load the association. This one is not needed since the JPA `FetchType.EAGER` offers the same behavior.

NO_PROXY
: This option will fetch the association lazily while returning real entity object.

PROXY
: This option will fetch the association lazily while returning a proxy instead.

24.2.47. `@ListIndexBase`

The `@ListIndexBase` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ListIndexBase.html) annotation is used to specify the start value for a list index, as stored in the database.

By default, `List` indexes are stored starting at zero. Generally used in conjunction with `@OrderColumn`.

24.2.48. `@Loader`

The `@Loader` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Loader.html) annotation is used to override the default `SELECT` query used for loading an entity loading.

See the Custom CRUD mapping section for more info.

24.2.49. `@ManyToAny`

The `@ManyToAny` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ManyToAny.html) annotation is used to specify a many-to-one association when the target type is dynamically resolved.

See the `@ManyToAny` mapping section for more info.

24.2.50. `@MapKeyType`

The `@MapKeyType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/MapKeyType.html) annotation is used to specify the map key type.

24.2.51. `@MetaValue`

The `@MetaValue` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/MetaValue.html) annotation is used by the `@AnyMetaDef` annotation to specify the association between a given discriminator value and an entity type.

See the `@Any` mapping section for more info.

24.2.52. `@NamedNativeQueries`

The `@NamedNativeQueries` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NamedNativeQueries.html) annotation is used to group multiple `@NamedNativeQuery` annotations.

24.2.53. `@NamedNativeQuery`

The `@NamedNativeQuery` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NamedNativeQuery.html) annotation extends the JPA `@NamedNativeQuery` with Hibernate specific features.

24.2.54. `@NamedQueries`

The `@NamedQueries` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NamedQueries.html) annotation is used to group multiple `@NamedQuery` annotations.

24.2.55. `@NamedQuery`

The `@NamedQuery` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NamedQuery.html) annotation extends the JPA `@NamedQuery` with Hibernate specific features.

24.2.56. `@Nationalized`

The `@Nationalized` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Nationalized.html) annotation is used to specify that the current annotated attribute is a character type (e.g. `String`, `Character`, `Clob`) that is stored in a nationalized column type (`NVARCHAR`, `NCHAR`, `NCLOB`).

See the `@Nationalized` mapping section for more info.

24.2.57. `@NaturalId`

The `@NaturalId` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NaturalId.html) annotation is used to specify that the current annotated attribute is part of the natural id of the entity.

See the Natural Ids section for more info.

24.2.58. `@NaturalIdCache`

The `@NaturalIdCache` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NaturalIdCache.html) annotation is used to specify that the natural id values associated with the annotated entity should be stored in the second-level cache.

See the `@NaturalIdCache` mapping section for more info.

24.2.59. `@NotFound`

The `@NotFound` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NotFound.html) annotation is used to specify the `NotFoundAction` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NotFoundAction.html) strategy for when an element is not found in a given association.

The `NotFoundAction` defines with two possibilities:

`EXCEPTION`
: An exception is thrown when an element is not found (default and recommended).

`IGNORE`
: Ignore the element when not found in the database.

24.2.60. `@OnDelete`

The `@OnDelete` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OnDelete.html) annotation is used to specify the delete strategy employed by the current annotated collection, array or joined subclasses. This annotation is used by the automated schema generation tool to generated the appropriate FOREIGN KEY DDL cascade directive.

The two possible strategies are defined by the `OnDeleteAction` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OnDeleteAction.html) enumeration:

CASCADE
: Use the database FOREIGN KEY cascade capabilities.

NO_ACTION
: Take no action.

24.2.61. `@OptimisticLock`

The `@OptimisticLock` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLock.html) annotation is used to specify if the current annotated attribute will trigger an entity version increment upon being modified.

24.2.62. `@OptimisticLocking`

The `@OptimisticLocking` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLocking.html) annotation is used to specify the current annotated an entity optimistic locking strategy.

The four possible strategies are defined by the `OptimisticLockType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLockType.html) enumeration:

NONE
: The implicit optimistic locking mechanism is disabled.

VERSION
: The implicit optimistic locking mechanism is using a dedicated version column.

ALL
: The implicit optimistic locking mechanism is using **all** attributes as part of an expanded WHERE clause restriction for the `UPDATE` and `DELETE` SQL statements.

DIRTY
: The implicit optimistic locking mechanism is using the **dirty** attributes (the attributes that were modified) as part of an expanded WHERE clause restriction for the `UPDATE` and `DELETE` SQL statements.

See the Versionless optimistic locking section for more info.

24.2.63. `@OrderBy`

The `@OrderBy` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OrderBy.html) annotation is used to specify a **SQL** ordering directive for sorting the current annotated collection.

It differs from the JPA `@OrderBy` annotation because the JPA annotation expects a JPQL order-by fragment, not an SQL directive.

24.2.64. `@ParamDef`

The `@ParamDef` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ParamDef.html) annotation is used in conjunction with `@FilterDef` so that the Hibernate Filter can be customized with runtime-provided parameter values.

See the Filter mapping section for more info.

24.2.65. `@Parameter`

The `@Parameter` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Parameter.html) annotation is generic parameter (basically a key/value combination) tused to parametrize other annotations, like `@CollectionType`, `@GenericGenerator`, and `@Type`, `@TypeDef`.

24.2.66. `@Parent`

The `@Parent` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Parent.html) annotation is used to specify that the current annotated embeddable attribute references back the owning entity.

24.2.67. `@Persister`

The `@Persister` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Persister.html) annotation is used to specify a custom entity or collection persister.

For entities, the custom persister must implement the `EntityPersister` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/persister/entity/EntityPersister.html) interface.

For collections, the custom persister must implement the `CollectionPersister` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/persister/collection/CollectionPersister.html) interface.

24.2.68. `@Polymorphism`

The `@Polymorphism` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Polymorphism.html) annotation is used to define the `PolymorphismType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/PolymorphismType.html) Hibernate will apply to entity hierarchies.

There are two possible `PolymorphismType` options:

EXPLICIT
: The current annotated entity is retrieved only if explicitly asked.

IMPLICIT
: The current annotated entity is retrieved if any of its super entity are retrieved. This is the default option.

24.2.69. `@Proxy`

The `@Proxy` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Proxy.html) annotation is used to specify a custom Proxy implementation for the current annotated entity.

24.2.70. `@RowId`

The `@RowId` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/RowId.html) annotation is used to specify the database column used as a `ROWID` pseudocolumn. For instance, Oracle defines the `ROWID` pseudocolumn (https://docs.oracle.com/cd/B19306_01/server.102/b14200/pseudocolumns008.htm) which provides the address of every table row.

According to Oracle documentation, `ROWID` is the fastest way to access a single row from a table.

24.2.71. `@SelectBeforeUpdate`

The `@SelectBeforeUpdate` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SelectBeforeUpdate.html) annotation is used to specify that the current annotated entity state be selected from the database when determining whether to perform an update when the detached entity is reattached.

See the `OptimisticLockType.DIRTY` mapping section for more info on how `@SelectBeforeUpdate` works.

24.2.72. ~~`@Sort`~~

The ~~`@Sort`~~ (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Sort.html) annotation is deprecated. Use the Hibernate specific `@SortComparator` or `@SortNatural` annotations instead.

24.2.73. `@SortComparator`

The `@SortComparator` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SortComparator.html) annotation is used to specify a `Comparator` for sorting the `Set` / `Map` in-memory.

See the `@SortComparator` mapping section for more info.

24.2.74. `@SortNatural`

The `@SortNatural` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SortNatural.html) annotation is used to specify that the `Set` / `Map` should be sorted using natural sorting.

See the `@SortNatural` mapping section for more info.

24.2.75. `@Source`

The `@Source` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Source.html) annotation is used in conjunction with a `@Version` timestamp entity attribute indicating the `SourceType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SourceType.html) of the timestamp value.

The `SourceType` offers two options:

DB
: Get the timestamp from the database.

VM
: Get the timestamp from the current JVM.

24.2.76. `@SQLDelete`

The `@SQLDelete` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SQLDelete.html) annotation is used to specify a custom SQL `DELETE` statement for the current annotated entity or collection.

See the Custom CRUD mapping section for more info.

24.2.77. `@SQLDeleteAll`

The `@SQLDeleteAll` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SQLDeleteAll.html) annotation is used to specify a custom SQL `DELETE` statement when removing all elements of the current annotated collection.

See the Custom CRUD mapping section for more info.

24.2.78. `@SqlFragmentAlias`

The `@SqlFragmentAlias` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SqlFragmentAlias.html) annotation is used to specify an alias for a Hibernate `@Filter`.

The alias (e.g. `myAlias`) can then be used in the `@Filter condition` clause using the `{alias}` (e.g. `{myAlias}`) placeholder.

24.2.79. `@SQLInsert`

The `@SQLInsert` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SQLInsert.html) annotation is used to specify a custom SQL `INSERT` statement for the current annotated entity or collection.

See the Custom CRUD mapping section for more info.

24.2.80. `@SQLUpdate`

The `@SQLUpdate` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/SQLUpdate.html) annotation is used to specify a custom SQL `UPDATE` statement for the current annotated entity or collection.

See the Custom CRUD mapping section for more info.

24.2.81. `@Subselect`

The `@Subselect` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Subselect.html) annotation is used to specify an immutable and read-only entity using a custom SQL `SELECT` statement.

24.2.82. `@Synchronize`

The `@Synchronize` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Synchronize.html) annotation is usually used in conjunction with the `@Subselect` annotation to specify the list of database tables used by the `@Subselect` SQL query.

With this information in place, Hibernate will properly trigger an entity flush whenever a query targeting the `@Subselect` entity is to be executed while the Persistence Context has scheduled some insert/update/delete actions against the database tables used by the `@Subselect` SQL query.

Therefore, the `@Synchronize` annotation prevents the derived entity from returning stale data when executing entity queries against the `@Subselect` entity.

24.2.83. `@Table`

The `@Table` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Table.html) annotation is used to specify additional information to a JPA `@Table` annotation, like custom `INSERT`, `UPDATE` or `DELETE` statements or a specific `FetchMode` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/FetchMode.html).

24.2.84. `@Tables`

The `@Tables` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Tables.html) annotation is used to group multiple `@Table` annotations.

24.2.85. `@Target`

The `@Target` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Target.html) annotation is used to specify an explicit target implementation when the current annotated association is using an interface type.

24.2.86. `@Tuplizer`

The `@Tuplizer` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Tuplizer.html) annotation is used to specify a custom tuplizer for the current annotated entity or embeddable.

For entities, the tupelizer must implement the `EntityTuplizer` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tuple/entity/EntityTuplizer.html) interface.

For embeddables, the tupelizer must implement the `ComponentTuplizer` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/tuple/component/ComponentTuplizer.html) interface.

24.2.87. `@Tuplizers`

The `@Tuplizers` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Tuplizers.html) annotation is used to group multiple `@Tuplizer` annotations.

24.2.88. `@Type`

The `@Type` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Type.html) annotation is used to specify the Hibernate `@Type` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/type/Type.html) used by the current annotated basic attribute.

See the `@Type` mapping section for more info.

24.2.89. `@TypeDef`

The `@TypeDef` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/TypeDef.html) annotation is used to specify a `@Type` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/type/Type.html) definition which can later be reused for multiple basic attribute mappings.

24.2.90. `@TypeDefs`

The `@TypeDefs` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/TypeDefs.html) annotation is used to group multiple `@TypeDef` annotations.

24.2.91. `@UpdateTimestamp`

The `@UpdateTimestamp` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/UpdateTimestamp.html) annotation is used to specify that the current annotated timestamp attribute should be updated with the current JVM timestamp whenever the owning entity gets modified.

- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

24.2.92. `@ValueGenerationType`

The `@ValueGenerationType` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/ValueGenerationType.html) annotation is used to specify that the current annotation type should be used as a generator annotation type.

See the `@ValueGenerationType` mapping section for more info.

24.2.93. `@Where`

The `@Where` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/Where.html) annotation is used to specify a custom SQL `WHERE` clause used when fetching an entity or a collection.

See the `@Where` mapping section for more info.

24.2.94. `@WhereJoinTable`

The `@WhereJoinTable` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/WhereJoinTable.html) annotation is used to specify a custom SQL `WHERE` clause used when fetching a join collection table.

25. Performance Tuning and Best Practices

Every enterprise system is unique. However, having a very efficient data access layer is a common requirement for many enterprise applications. Hibernate comes with a great variety of features that can help you tune the data access layer.

25.1. Schema management

Although Hibernate provides the `update` option for the `hibernate.hbm2ddl.auto` configuration property, this feature is not suitable for a production environment.

An automated schema migration tool (e.g. Flyway (https://flywaydb.org/), Liquibase (http://www.liquibase.org/)) allows you to use any database-specific DDL feature (e.g. Rules, Triggers, Partitioned Tables). Every migration should have an associated script, which is stored on the Version Control System, along with the application source code.

When the application is deployed on a production-like QA environment, and the deploy worked as expected, then pushing the deploy to a production environment should be straightforward since the latest schema migration was already tested.

> You should always use an automatic schema migration tool and have all the migration scripts stored in the Version Control System.

25.2. Logging

Whenever you're using a framework that generates SQL statements on your behalf, you have to ensure that the generated statements are the ones that you intended in the first place.

There are several alternatives to logging statements. You can log statements by configuring the underlying logging framework. For Log4j, you can use the following appenders:

JAVA

```
### log just the SQL
log4j.logger.org.hibernate.SQL=debug

### log JDBC bind parameters ###
log4j.logger.org.hibernate.type=trace
log4j.logger.org.hibernate.type.descriptor.sql=trace
```

However, there are some other alternatives like using datasource-proxy or p6spy. The advantage of using a JDBC `Driver` or `DataSource` Proxy is that you can go beyond simple SQL logging:

- statement execution time
- JDBC batching logging
- database connection monitoring (https://github.com/vladmihalcea/flexy-pool)

Another advantage of using a `DataSource` proxy is that you can assert the number of executed statements at test time. This way, you can have the integration tests fail when a N+1 query issue is automatically detected.

While simple statement logging is fine, using datasource-proxy (https://github.com/ttddyy/datasource-proxy) or p6spy (https://github.com/p6spy/p6spy) is even better.

25.3. JDBC batching

JDBC allows us to batch multiple SQL statements and to send them to the database server into a single request. This saves database roundtrips, and so it reduces response time significantly (https://leanpub.com/high-performance-java-persistence/read#jdbc-batch-updates).

Not only `INSERT` and `UPDATE` statements, but even `DELETE` statements can be batched as well. For `INSERT` and `UPDATE` statements, make sure that you have all the right configuration properties in place, like ordering inserts and updates and activating batching for versioned data. Check out this article for more details on this topic.

For `DELETE` statements, there is no option to order parent and child statements, so cascading can interfere with the JDBC batching process.

Unlike any other framework which doesn't automate SQL statement generation, Hibernate makes it very easy to activate JDBC-level batching as indicated in the Batching chapter.

25.4. Mapping

Choosing the right mappings is very important for a high-performance data access layer. From the identifier generators to associations, there are many options to choose from, yet not all choices are equal from a performance perspective.

25.4.1. Identifiers

When it comes to identifiers, you can either choose a natural id or a synthetic key.

For natural identifiers, the **assigned** identifier generator is the right choice.

For synthetic keys, the application developer can either choose a randomly generates fixed-size sequence (e.g. UUID) or a natural identifier. Natural identifiers are very practical, being more compact than their UUID counterparts, so there are multiple generators to choose from:

- `IDENTITY`
- `SEQUENCE`
- `TABLE`

Although the `TABLE` generator addresses the portability concern, in reality, it performs poorly because it requires emulating a database sequence using a separate transaction and row-level locks. For this reason, the choice is usually between `IDENTITY` and `SEQUENCE`.

> If the underlying database supports sequences, you should always use them for your Hibernate entity identifiers.
>
> Only if the relational database does not support sequences (e.g. MySQL 5.7), you should use the `IDENTITY` generators. However, you should keep in mind that the `IDENTITY` generators disables JDBC batching for `INSERT` statements.

If you're using the `SEQUENCE` generator, then you should be using the enhanced identifier generators that were enabled by default in Hibernate 5. The **pooled** and the **pooled-lo** optimizers are very useful to reduce the number of database roundtrips when writing multiple entities per database transaction.

25.4.2. Associations

JPA offers four entity association types:

- `@ManyToOne`
- `@OneToOne`
- `@OneToMany`
- `@ManyToMany`

And an `@ElementCollection` for collections of embeddables.

Because object associations can be bidirectional, there are many possible combinations of associations. However, not every possible association type is efficient from a database perspective.

> The closer the association mapping is to the underlying database relationship, the better it will perform.
>
> On the other hand, the more exotic the association mapping, the better the chance of being inefficient.

Therefore, the `@ManyToOne` and the `@OneToOne` child-side association are best to represent a `FOREIGN KEY` relationship.

The parent-side `@OneToOne` association requires bytecode enhancement so that the association can be loaded lazily. Otherwise, the parent-side is always fetched even if the association is marked with `FetchType.LAZY`.

For this reason, it's best to map `@OneToOne` association using `@MapsId` so that the `PRIMARY KEY` is shared between the child and the parent entities. When using `@MapsId`, the parent-side becomes redundant since the child-entity can be easily fetched using the parent entity identifier.

For collections, the association can be either:

- unidirectional
- bidirectional

For unidirectional collections, `Sets` are the best choice because they generate the most efficient SQL statements. Unidirectional `Lists` are less efficient than a `@ManyToOne` association.

Bidirectional associations are usually a better choice because the `@ManyToOne` side controls the association.

Embeddable collections (`@ElementCollection`) are unidirectional associations, hence `Sets` are the most efficient, followed by ordered `Lists`, whereas bags (unordered `Lists`) are the least efficient.

The `@ManyToMany` annotation is rarely a good choice because it treats both sides as unidirectional associations.

For this reason, it's much better to map the link table as depicted in the Bidirectional many-to-many with link entity lifecycle section. Each `FOREIGN KEY` column will be mapped as a `@ManyToOne` association. On each parent-side, a bidirectional `@OneToMany` association is going to map to the aforementioned `@ManyToOne` relationship in the link entity.

> Just because you have support for collections, it does not mean that you have to turn any one-to-many database relationship into a collection.
>
> Sometimes, a `@ManyToOne` association is sufficient, and the collection can be simply replaced by an entity query which is easier to paginate or filter.

25.5. Inheritance

JPA offers `SINGLE_TABLE`, `JOINED`, and `TABLE_PER_CLASS` to deal with inheritance mapping, and each of these strategies has advantages and disadvantages.

- `SINGLE_TABLE` performs the best in terms of executed SQL statements. However, you cannot use `NOT NULL` constraints on the column-level. You can still use triggers and rules to enforce such constraints, but it's not as straightforward.
- `JOINED` addresses the data integrity concerns because every subclass is associated with a different table. Polymorphic queries or `@OneToMany` base class associations don't perform very well with this strategy. However, polymorphic @ManyToOne` associations are fine, and they can provide a lot of value.
- `TABLE_PER_CLASS` should be avoided since it does not render efficient SQL statements.

25.6. Fetching

> Fetching too much data is the number one performance issue for the vast majority of JPA applications.

Hibernate supports both entity queries (JPQL/HQL and Criteria API) and native SQL statements. Entity queries are useful only if you need to modify the fetched entities, therefore benefiting from the automatic dirty checking mechanism.

For read-only transactions, you should fetch DTO projections because they allow you to select just as many columns as you need to fulfill a certain business use case. This has many benefits like reducing the load on the currently running Persistence Context because DTO projections don't need to be managed.

25.6.1. Fetching associations

Related to associations, there are two major fetch strategies:

- `EAGER`
- `LAZY`

`EAGER` fetching is almost always a bad choice.

> Prior to JPA, Hibernate used to have all associations as `LAZY` by default. However, when JPA 1.0 specification emerged, it was thought that not all providers would use Proxies. Hence, the `@ManyToOne` and the `@OneToOne` associations are now `EAGER` by default.
>
> The `EAGER` fetching strategy cannot be overwritten on a per query basis, so the association is always going to be retrieved even if you don't need it. More, if you forget to `JOIN FETCH` an `EAGER` association in a JPQL query, Hibernate will initialize it with a secondary statement, which in turn can lead to N+1 query issues.

So, `EAGER` fetching is to be avoided. For this reason, it's better if all associations are marked as `LAZY` by default.

However, `LAZY` associations must be initialized prior to being accessed. Otherwise, a `LazyInitializationException` is thrown. There are good and bad ways to treat the `LazyInitializationException` .

The best way to deal with `LazyInitializationException` is to fetch all the required associations prior to closing the Persistence Context. The `JOIN FETCH` directive is good for `@ManyToOne` and `OneToOne` associations, and for at most one collection (e.g. `@OneToMany` or `@ManyToMany`). If you need to fetch multiple collections, to avoid a Cartesian Product, you should use secondary queries which are triggered either by navigating the `LAZY` association or by calling `Hibernate#initialize(proxy)` method.

25.7. Caching

Hibernate has two caching layers:

- the first-level cache (Persistence Context) which is a application-level repeatable reads.
- the second-level cache which, unlike application-level caches, it doesn't store entity aggregates but normalized dehydrated entity entries.

The first-level cache is not a caching solution "per se", being more useful for ensuring `READ COMMITTED` isolation level.

While the first-level cache is short lived, being cleared when the underlying `EntityManager` is closed, the second-level cache is tied to an `EntityManagerFactory`. Some second-level caching providers offer support for clusters. Therefore, a node needs only to store a subset of the whole cached data.

Although the second-level cache can reduce transaction response time since entities are retrieved from the cache rather than from the database, there are other options to achieve the same goal, and you should consider these alternatives prior to jumping to a second-level cache layer:

- tuning the underlying database cache so that the working set fits into memory, therefore reducing Disk I/O traffic.
- optimizing database statements through JDBC batching, statement caching, indexing can reduce the average response time, therefore increasing throughput as well.
- database replication is also a very valuable option to increase read-only transaction throughput

After properly tuning the database, to further reduce the average response time and increase the system throughput, application-level caching becomes inevitable.

Topically, a key-value application-level cache like Memcached (https://memcached.org/) or Redis (http://redis.io/) is a common choice to store data aggregates. If you can duplicate all data in the key-value store, you have the option of taking down the database system for maintenance without completely loosing availability since read-only traffic can still be served from the cache.

One of the main challenges of using an application-level cache is ensuring data consistency across entity aggregates. That's where the second-level cache comes to the rescue. Being tightly integrated with Hibernate, the second-level cache can provide better data consistency since entries are cached in a normalized fashion, just like in a relational database. Changing a parent entity only requires a single entry cache update, as opposed to cache entry invalidation cascading in key-value stores.

The second-level cache provides four cache concurrency strategies:

- `READ_ONLY`
- `NONSTRICT_READ_WRITE`
- `READ_WRITE`
- `TRANSACTIONAL`

`READ_WRITE` is a very good default concurrency strategy since it provides strong consistency guarantees without compromising throughput. The `TRANSACTIONAL` concurrency strategy uses JTA. Hence, it's more suitable when entities are frequently modified.

Both `READ_WRITE` and `TRANSACTIONAL` use write-through caching, while `NONSTRICT_READ_WRITE` is a read-through caching strategy. For this reason, `NONSTRICT_READ_WRITE` is not very suitable if entities are changed frequently.

When using clustering, the second-level cache entries are spread across multiple nodes. When using Infinispan distributed cache (http://blog.infinispan.org/2015/10/hibernate-second-level-cache.html), only `READ_WRITE` and `NONSTRICT_READ_WRITE` are available for read-write caches. Bear in mind that `NONSTRICT_READ_WRITE` offers a weaker consistency guarantee since stale updates are possible.

> For more about Hibernate Performance Tuning, check out the High-Performance Hibernate (https://www.youtube.com/watch?v=BTdTEe9QL5k&t=1s) presentation from Devoxx France.

26. Legacy Bootstrapping

The legacy way to bootstrap a SessionFactory is via the `org.hibernate.cfg.Configuration` object. `Configuration` represents, essentially, a single point for specifying all aspects of building the `SessionFactory` : everything from settings, to mappings, to strategies, etc. I like to think of `Configuration` as a big pot to which we add a bunch of stuff (mappings, settings, etc) and from which we eventually get a `SessionFactory.`

> There are some significant draw backs to this approach which led to its deprecation and the development of the new approach, which is discussed in Native Bootstrapping. `Configuration` is semi-deprecated but still available for use, in a limited form that eliminates these drawbacks. "Under the covers", `Configuration` uses the new bootstrapping code, so the things available there as also available here in terms of auto-discovery.

You can obtain the `Configuration` by instantiating it directly. You then specify mapping metadata (XML mapping documents, annotated classes) that describe your applications object model and its mapping to a SQL database.

JAVA

```
Configuration cfg = new Configuration()
    // addResource does a classpath resource lookup
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml")

    // calls addResource using "/org/hibernate/auction/User.hbm.xml"
    .addClass(`org.hibernate.auction.User.class`)

    // parses Address class for mapping annotations
    .addAnnotatedClass( Address.class )

    // reads package-level (package-info.class) annotations in the named package
    .addPackage( "org.hibernate.auction" )

    .setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

There are other ways to specify Configuration information, including:

- Place a file named hibernate.properties in a root directory of the classpath
- Pass an instance of java.util.Properties to `Configuration#setProperties`
- Via a `hibernate.cfg.xml` file
- System properties using java `-Dproperty=value`

27. Migration

Mapping Configuration methods to the corresponding methods in the new APIs..

`Configuration#addFile`	`Configuration#addFile`
`Configuration#add(XmlDocument)`	`Configuration#add(XmlDocument)`
`Configuration#addXML`	`Configuration#addXML`
`Configuration#addCacheableFile`	`Configuration#addCacheableFile`
`Configuration#addURL`	`Configuration#addURL`
`Configuration#addInputStream`	`Configuration#addInputStream`

Configuration#addResource	Configuration#addResource
Configuration#addClass	Configuration#addClass
Configuration#addAnnotatedClass	Configuration#addAnnotatedClass
Configuration#addPackage	Configuration#addPackage
Configuration#addJar	Configuration#addJar
Configuration#addDirectory	Configuration#addDirectory
Configuration#registerTypeContributor	Configuration#registerTypeContributor
Configuration#registerTypeOverride	Configuration#registerTypeOverride
Configuration#setProperty	Configuration#setProperty
Configuration#setProperties	Configuration#setProperties
Configuration#addProperties	Configuration#addProperties
Configuration#setNamingStrategy	Configuration#setNamingStrategy
Configuration#setImplicitNamingStrategy	Configuration#setImplicitNamingStrategy
Configuration#setPhysicalNamingStrategy	Configuration#setPhysicalNamingStrategy
Configuration#configure	Configuration#configure
Configuration#setInterceptor	Configuration#setInterceptor
Configuration#setEntityNotFoundDelegate	Configuration#setEntityNotFoundDelegate
Configuration#setSessionFactoryObserver	Configuration#setSessionFactoryObserver
Configuration#setCurrentTenantIdentifierResolver	Configuration#setCurrentTenantIdentifierResolver

28. Legacy Domain Model

Example 537. Declaring a version property in `hbm.xml`

```xml
<!--
  ~ Hibernate, Relational Persistence for Idiomatic Java
  ~
  ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.
  ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
  -->
<version
    column="version_column"
    name="propertyName"
    type="typename"
    access="field|property|ClassName"
    unsaved-value="null|negative|undefined"
    generated="never|always"
    insert="true|false"
    node="element-name|@attribute-name|element/@attribute|."
/>
```

column	The name of the column holding the version number. Optional, defaults to the property name.
name	The name of a property of the persistent class.
type	The type of the version number. Optional, defaults to `integer`.
access	Hibernate's strategy for accessing the property value. Optional, defaults to `property`.
unsaved-value	Indicates that an instance is newly instantiated and thus unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value, `undefined`, indicates that the identifier property value should be used. Optional.
generated	Indicates that the version property value is generated by the database. Optional, defaults to `never`.
insert	Whether or not to include the `version` column in SQL `insert` statements. Defaults to `true`, but you can set it to `false` if the database column is defined with a default value of `0`.

Example 538. The timestamp element in `hbm.xml`

```xml
<!--
  ~ Hibernate, Relational Persistence for Idiomatic Java
  ~
  ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.
  ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
  -->
<timestamp
    column="timestamp_column"
    name="propertyName"
    access="field|property|ClassName"
    unsaved-value="null|undefined"
    source="vm|db"
    generated="never|always"
    node="element-name|@attribute-name|element/@attribute|."
/>
```

column	The name of the column which holds the timestamp. Optional, defaults to the property name
name	The name of a JavaBeans style property of Java type `Date` or `Timestamp` of the persistent class.
access	The strategy Hibernate uses to access the property value. Optional, defaults to `property`.
unsaved-value	A version property which indicates than instance is newly instantiated, and unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value of `undefined` indicates that Hibernate uses the identifier property value.
source	Whether Hibernate retrieves the timestamp from the database or the current JVM. Database-based timestamps incur an overhead because Hibernate needs to query the database each time to determine the incremental next value. However, database-derived timestamps are safer to use in a clustered environment. Not all database dialects are known to support the retrieval of the database's current timestamp. Others may also be unsafe for locking because of lack of precision.
generated	Whether the timestamp property value is generated by the database. Optional, defaults to `never`.

29. Legacy Hibernate Criteria Queries

> This appendix covers the legacy Hibernate `org.hibernate.Criteria` API, which should be considered deprecated.
>
> New development should focus on the JPA javax.persistence.criteria.CriteriaQuery API. Eventually, Hibernate-specific criteria features will be ported as extensions to the JPA `javax.persistence.criteria.CriteriaQuery`. For details on the JPA APIs, see Criteria.

Hibernate features an intuitive, extensible criteria query API.

29.1. Creating a `Criteria` instance

The interface `org.hibernate.Criteria` represents a query against a particular persistent class. The `Session` is a factory for `Criteria` instances.

JAVA

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

29.2. JPA vs Hibernate entity name

When using the `Session#createCriteria(String entityName)` or `StatelessSession#createCriteria(String entityName)` (https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/SharedSessionContract.html#createCriteria-java.lang.String-), the **entityName** means the fully-qualified name of the underlying entity and not the name denoted by the `name` attribute of the JPA `@Entity` annotation.

Considering you have the following entity:

JAVA

```
@Entity(name = "ApplicationEvent")
public static class Event {

    @Id
    private Long id;

    private String name;
}
```

If you provide the JPA entity name to a legacy Criteria query:

JAVA
```
List<Event> events =
    entityManager.unwrap( Session.class )
.createCriteria( "ApplicationEvent" )
.list();
```

Hibernate is going to throw the following `MappingException`:

BASH
```
org.hibernate.MappingException: Unknown entity: ApplicationEvent
```

On the other hand, the Hibernate entity name (the fully qualified class name) works just fine:

JAVA
```
List<Event> events =
    entityManager.unwrap( Session.class )
.createCriteria( Event.class.getName() )
.list();
```

For more about this topic, check out the HHH-2597 (https://hibernate.atlassian.net/browse/HHH-2597) JIRA issue.

29.3. Narrowing the result set

An individual query criterion is an instance of the interface `org.hibernate.criterion.Criterion`. The class `org.hibernate.criterion.Restrictions` defines factory methods for obtaining certain built-in `Criterion` types.

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Restrictions can be grouped logically.

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

There are a range of built-in criterion types (`Restrictions` subclasses). One of the most useful `Restrictions` allows you to specify SQL directly.

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

The `{alias}` placeholder will be replaced by the row alias of the queried entity.

You can also obtain a criterion from a `Property` instance. You can create a `Property` by calling `Property.forName()`:

JAVA
```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

29.4. Ordering the results

You can order the results using `org.hibernate.criterion.Order`.

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name").nulls(NullPrecedence.LAST) )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

29.5. Associations

By navigating associations using `createCriteria()` you can specify constraints upon related entities:

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

The second `createCriteria()` returns a new instance of `Criteria` that refers to the elements of the `kittens` collection.

There is also an alternate form that is useful in certain circumstances:

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` does not create a new instance of `Criteria`.)

The kittens collections held by the `Cat` instances returned by the previous two queries are *not* pre-filtered by the criteria. If you want to retrieve just the kittens that match the criteria, you must use a `ResultTransformer`.

JAVA
```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

Additionally, you may manipulate the result set using a left outer join:

```
List cats = session.createCriteria( Cat.class )
    .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name", "good%") )
    .addOrder(Order.asc("mt.age"))
    .list();
```

This will return all of the `Cat`s with a mate whose name starts with "good" ordered by their mate's age, and all cats who do not have a mate. This is useful when there is a need to order or limit in the database prior to returning complex/large result sets, and removes many instances where multiple queries would have to be performed and the results unioned by java in memory.

Without this feature, first all of the cats without a mate would need to be loaded in one query.

A second query would need to retrieve the cats with mates who's name started with "good" sorted by the mates age.

Thirdly, in memory; the lists would need to be joined manually.

29.6. Dynamic association fetching

You can specify association fetching semantics at runtime using `setFetchMode()`.

JAVA

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join.

29.7. Components

To add a restriction against a property of an embedded component, the component property name should be prepended to the property name when creating the `Restriction`. The criteria object should be created on the owning entity, and cannot be created on the component itself. For example, suppose the `Cat` has a component property `fullName` with sub-properties `firstName` and `lastName`:

```
List cats = session.createCriteria(Cat.class)
    .add(Restrictions.eq("fullName.lastName", "Cattington"))
    .list();
```

Note: this does not apply when querying collections of components, for that see below Collections

29.8. Collections

When using criteria against collections, there are two distinct cases. One is if the collection contains entities (eg. `<one-to-many/>` or `<many-to-many/>`) or components (`<composite-element/>`), and the second is if the collection contains scalar values (`<element/>`). In the first case, the syntax is as given above in the section Associations where we restrict the `kittens` collection.

Essentially we create a `Criteria` object against the collection property and restrict the entity or component properties using that instance.

For querying a collection of basic values, we still create the `Criteria` object against the collection, but to reference the value, we use the special property "elements". For an indexed collection, we can also reference the index property using the special property "indices".

```
List cats = session.createCriteria(Cat.class)
    .createCriteria("nickNames")
    .add(Restrictions.eq("elements", "BadBoy"))
    .list();
```

29.9. Example queries

The class `org.hibernate.criterion.Example` allows you to construct a query criterion from a given instance.

JAVA

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Version properties, identifiers and associations are ignored. By default, null valued properties are excluded.

You can adjust how the `Example` is applied.

JAVA

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

You can even use examples to place criteria upon associated objects.

JAVA

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

29.10. Projections, aggregation and grouping

The class `org.hibernate.criterion.Projections` is a factory for `Projection` instances. You can apply a projection to a query by calling `setProjection()`.

JAVA

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

JAVA

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

There is no explicit "group by" necessary in a criteria query. Certain projection types are defined to be *grouping projections*, which also appear in the SQL `group by` clause.

An alias can be assigned to a projection so that the projected value can be referred to in restrictions or orderings. Here are two different ways to do this:

JAVA

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

JAVA

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

The `alias()` and `as()` methods simply wrap a projection instance in another, aliased, instance of `Projection`. As a shortcut, you can assign an alias when you add the projection to a projection list:

JAVA

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```java
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

You can also use `Property.forName()` to express projections:

```java
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```java
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

29.11. Detached queries and subqueries

The `DetachedCriteria` class allows you to create a query outside the scope of a session and then execute it using an arbitrary `Session`.

```java
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

A `DetachedCriteria` can also be used to express a subquery. `Criterion` instances involving subqueries can be obtained via `Subqueries` or `Property`.

```java
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight").gt(avgWeight) )
    .list();
```

```java
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Correlated subqueries are also possible:

```java
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight").gt(avgWeightForSex) )
    .list();
```

Example of multi-column restriction based on a subquery:

```java
DetachedCriteria sizeQuery = DetachedCriteria.forClass( Man.class )
    .setProjection( Projections.projectionList().add( Projections.property( "weight" ) )
                                                .add( Projections.property( "height" ) ) )
    .add( Restrictions.eq( "name", "John" ) );
session.createCriteria( Woman.class )
    .add( Subqueries.propertiesEq( new String[] { "weight", "height" }, sizeQuery ) )
    .list();
```

29.12. Queries by natural identifier

For most queries, including criteria queries, the query cache is not efficient because query cache invalidation occurs too frequently. However, there is a special kind of query where you can optimize the cache invalidation algorithm: lookups by a constant natural key. In some applications, this kind of query occurs frequently. The Criteria API provides special provision for this use case.

First, map the natural key of your entity using `<natural-id>` and enable use of the second-level cache.

XML

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

This functionality is not intended for use with entities with *mutable* natural keys.

Once you have enabled the Hibernate query cache, the `Restrictions.naturalId()` allows you to make use of the more efficient cache algorithm.

JAVA

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

30. Legacy Hibernate Native Queries

30.1. Legacy Named SQL queries

Named SQL queries can also be defined during mapping and called in exactly the same way as a named HQL query. In this case, you do *not* need to call `addEntity()` anymore.

Example 539. Named sql query using the `<sql-query>` mapping element

XML

```
<sql-query name = "persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Example 540. Execution of a named query

```java
List people = session
    .getNamedQuery( "persons" )
    .setParameter( "namePattern", namePattern )
    .setMaxResults( 50 )
    .list();
```

The `<return-join>` element is use to join associations and the `<load-collection>` element is used to define queries which initialize collections.

Example 541. Named sql query with association

```xml
<sql-query name = "personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

A named SQL query may return a scalar value. You must declare the column alias and Hibernate type using the `<return-scalar>` element:

Example 542. Named query returning a scalar

```xml
<sql-query name = "mySqlQuery">
    <return-scalar column = "name" type="string"/>
    <return-scalar column = "age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

You can externalize the resultset mapping information in a `<resultset>` element which will allow you to either reuse them across several named queries or through the `setResultSetMapping()` API.

Example 543. <resultset> mapping used to externalize mappinginformation

```xml
<resultset name = "personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name = "personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

XML

You can, alternatively, use the resultset mapping information in your hbm files directly in java code.

Example 544. Programmatically specifying the result mapping information

```java
List cats = session
    .createSQLQuery( "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id" )
    .setResultSetMapping("catAndKitten")
    .list();
```

JAVA

30.2. Legacy return-property to explicitly specify column/alias names

You can explicitly tell Hibernate what column aliases to use with `<return-property>`, instead of using the `{}` syntax to let Hibernate inject its own aliases. For example:

XML

```
<sql-query name = "mySqlQuery">
    <return alias = "person" class = "eg.Person">
        <return-property name = "name" column = "myName"/>
        <return-property name = "age" column = "myAge"/>
        <return-property name = "sex" column = "mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` also works with multiple columns. This solves a limitation with the `{}` syntax which cannot allow fine grained control of multi-column properties.

XML

```
<sql-query name = "organizationCurrentEmployments">
    <return alias = "emp" class = "Employment">
        <return-property name = "salary">
            <return-column name = "VALUE"/>
            <return-column name = "CURRENCY"/>
        </return-property>
        <return-property name = "endDate" column = "myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

In this example `<return-property>` was used in combination with the `{}` syntax for injection. This allows users to choose how they want to refer column and properties.

If your mapping has a discriminator you must use `<return-discriminator>` to specify the discriminator column.

30.3. Legacy stored procedures for querying

Hibernate provides support for queries via stored procedures and functions. Most of the following documentation is equivalent for both. The stored procedure/function must return a resultset as the first out-parameter to be able to work with Hibernate. An example of such a stored function in Oracle 9 and higher is as follows:

XML

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
        SELECT EMPLOYEE, EMPLOYER,
        STARTDATE, ENDDATE,
        REGIONCODE, EID, VALUE, CURRENCY
        FROM EMPLOYMENT;
    RETURN  st_cursor;
END;
```

To use this query in Hibernate you need to map it via a named query.

XML

```
<sql-query name = "selectAllEmployees_SP" callable = "true">
    <return alias="emp" class="Employment">
        <return-property name = "employee" column = "EMPLOYEE"/>
        <return-property name = "employer" column = "EMPLOYER"/>
        <return-property name = "startDate" column = "STARTDATE"/>
        <return-property name = "endDate" column = "ENDDATE"/>
        <return-property name = "regionCode" column = "REGIONCODE"/>
        <return-property name = "id" column = "EID"/>
        <return-property name = "salary">
            <return-column name = "VALUE"/>
            <return-column name = "CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Stored procedures currently only return scalars and entities. `<return-join>` and `<load-collection>` are not supported.

30.4. Legacy rules/limitations for using stored procedures

You cannot use stored procedures with Hibernate unless you follow some procedure/function rules. If they do not follow those rules they are not usable with Hibernate. If you still want to use these procedures you have to execute them via `session.doWork()`.

The rules are different for each database since database vendors have different stored procedure semantics/syntax.

Stored procedure queries cannot be paged with `setFirstResult()/setMaxResults()`.

The recommended call form is standard SQL92: `{ ? = call functionName(<parameters>) }` or `{ ? = call procedureName(<parameters>}`. Native call syntax is not supported.

For Oracle the following rules apply:

- A function must return a result set. The first parameter of a procedure must be an `OUT` that returns a result set. This is done by using a `SYS_REFCURSOR` type in Oracle 9 or 10. In Oracle you need to define a `REF CURSOR` type. See Oracle literature for further information.

For Sybase or MS SQL server the following rules apply:

- The procedure must return a result set. Note that since these servers can return multiple result sets and update counts, Hibernate will iterate the results and take the first result that is a result set as its return value. Everything else will be discarded.
- If you can enable `SET NOCOUNT ON` in your procedure it will probably be more efficient, but this is not a requirement.

30.5. Legacy custom SQL for create, update and delete

Hibernate can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or individual column level. This section describes statement overrides. For columns, see Column transformers: read and write expressions. The following example shows how to define custom SQL operations using annotations.

Example 545. Custom CRUD XML

```xml
<class name = "Person">
    <id name = "id">
        <generator class = "increment"/>
    </id>
    <property name = "name" not-null = "true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

> If you expect to call a store procedure, be sure to set the `callable` attribute to `true`, in annotations as well as in xml.

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 which is again available in annotations as well as in xml.

Last but not least, stored procedures are in most cases required to return the number of rows inserted, updated and deleted. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

Example 546. Stored procedures and their return value

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

30.6. Legacy custom SQL for loading

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in For columns, see Column transformers: read and write expressions or at the statement level. Here is an example of a statement level override:

XML

```
<sql-query name = "person">
    <return alias = "pers" class = "Person" lock-mod e= "upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

This is just a named query declaration, as discussed earlier. You can reference this named query in a class mapping:

XML

```
<class name = "Person">
    <id name = "id">
        <generator class = "increment"/>
    </id>
    <property name = "name" not-null = "true"/>
    <loader query-ref = "person"/>
</class>
```

This even works with stored procedures.

You can even define a query for collection loading:

XML

```
<set name = "employments" inverse = "true">
    <key/>
    <one-to-many class = "Employment"/>
    <loader query-ref = "employments"/>
</set>
```

XML

```
<sql-query name = "employments">
    <load-collection alias = "emp" role = "Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

You can also define an entity loader that loads a collection by join fetching:

XML

```
<sql-query name = "person">
    <return alias = "pers" class = "Person"/>
    <return-join alias = "emp" property = "pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

31. References

- [PoEAA] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Publishing Company. 2003.

- [JPwH] Christian Bauer & Gavin King. Java Persistence with Hibernate (http://www.manning.com/bauer2). Manning Publications Co. 2007.

Last updated 2017-02-16 11:14:38 +00:00