Hibernate Developer Guide

The Hibernate Team

The JBoss Visual Design Team

4.1.6.Final

2012-08-09

Table of Contents

List of Tables

List of Examples

Preface

Table of Contents

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 refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa). See http://en.wikipedia.org/wiki/Object-relational_mapping for a good high-level discussion.

Note

While having a strong background in SQL is not required to use Hibernate, having a basic understanding of the concepts can greatly help you understand Hibernate more fully and quickly. Probably the single best background is an understanding of data modeling principles. You might want to consider these resources as a good starting point:

- http://www.agiledata.org/essays/dataModeling101.html
- http://en.wikipedia.org/wiki/Data_modeling

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.

1. Get Involved

- Use Hibernate and report any bugs or issues you find. See http://hibernate.org/issuetracker.html for details.
- Try your hand at fixing some bugs or implementing enhancements. Again, see http://hibernate.org/issuetracker.html.
- Engage with the community using mailing lists, forums, IRC, or other ways listed at http://hibernate.org/community.html.
- 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.

2. Getting Started Guide

New users may want to first look through the *Hibernate Getting Started Guide* for basic information as well as tutorials. Even seasoned veterans may want to considering perusing the sections pertaining to build artifacts for any changes.

Chapter 1. Database access

Table of Contents

1.1. Connecting

Hibernate connects to databases on behalf of your application. It can connect through a variety of mechanisms, including:

- Stand-alone built-in connection pool
- `javax.sql.DataSource`
- Connection pools, including support for two different third-party opensource JDBC connection pools:
    - c3p0
    - proxool
- Application-supplied JDBC connections. This is not a recommended approach and exists for legacy reasons

Note

The built-in connection pool is not intended for production environments.

Hibernate obtains JDBC connections as needed though the `org.hibernate.service.jdbc.connections.spi.ConnectionProvider` interface which is a service contract. Applications may also supply their own `org.hibernate.service.jdbc.connections.spi.ConnectionProvider` implementation to define a custom approach for supplying connections to Hibernate (from a different connection pool implementation, for example).

1.1.1. Configuration

You can configure database connections using a properties file, an XML deployment descriptor or programmatically.

Example 1.1. `hibernate.properties` for a c3p0 connection pool

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQL82Dialect
```

Example 1.2. `hibernate.cfg.xml` for a connection to the bundled HSQL database

```
<?xml version='1.0' encoding='utf-8'?>


<hibernate-configuration

        xmlns="http://www.hibernate.org/xsd/hibernate-configuration"

        xsi:schemaLocation="http://www.hibernate.org/xsd/hibernate-configuration hibernate-configuration-4.0.xsd"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <session-factory>

    <!-- Database connection settings -->

    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>

    <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>

    <property name="connection.username">sa</property>

    <property name="connection.password"></property>


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

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

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

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

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

    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto">update</property>
    <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

1.1.1.1. Programatic configuration

An instance of object `org.hibernate.cfg.Configuration` represents an entire set of mappings of an application's Java types to an SQL database. The `org.hibernate.cfg.Configuration` builds an immutable `org.hibernate.SessionFactory`, and compiles the mappings from various XML mapping files. You can specify the mapping files directly, or Hibernate can find them for you.

Example 1.3. Specifying the mapping files directly

You can obtain a `org.hibernate.cfg.Configuration` instance by instantiating it directly and specifying XML mapping documents. If the mapping files are in the classpath, use method `addResource()`.

```
Configuration cfg = new Configuration()

    .addResource("Item.hbm.xml")

    .addResource("Bid.hbm.xml");
```

Example 1.4. Letting Hibernate find the mapping files for you

The `addClass()` method directs Hibernate to search the CLASSPATH for the mapping files, eliminating hard-coded file names. In the following example, it searches for `org/hibernate/auction/Item.hbm.xml` and `org/hibernate/auction/Bid.hbm.xml`.

```
Configuration cfg = new Configuration()

    .addClass(org.hibernate.auction.Item.class)

    .addClass(org.hibernate.auction.Bid.class);
```

Example 1.5. Specifying configuration properties

```
Configuration cfg = new Configuration()
```

```
    .addClass(org.hibernate.auction.Item.class)

    .addClass(org.hibernate.auction.Bid.class)

    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")

    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")

    .setProperty("hibernate.order_updates", "true");
```

Other ways to configure Hibernate programmatically

- Pass an instance of `java.util.Properties` to `Configuration.setProperties()`.
- Set System properties using **java -D*`property`*=*`value`***

1.1.2. Obtaining a JDBC connection

After you configure the Most important Hibernate JDBC properties, you can use method `openSession` of class `org.hibernate.SessionFactory` to open sessions. Sessions will obtain JDBC connections as needed based on the provided configuration.

Example 1.6. Specifying configuration properties

```
Session session = sessions.openSession();
```

Most important Hibernate JDBC properties

- hibernate.connection.driver_class
- hibernate.connection.url
- hibernate.connection.username
- hibernate.connection.password

- hibernate.connection.pool_size

All available Hibernate settings are defined as constants and discussed on the `org.hibernate.cfg.AvailableSettings` interface. See its source code or JavaDoc for details.

1.2. Connection pooling

Hibernate's internal connection pooling algorithm is rudimentary, and is provided for development and testing purposes. Use a third-party pool for best performance and stability. To use a third-party pool, replace the hibernate.connection.pool_size property with settings specific to your connection pool of choice. This disables Hibernate's internal connection pool.

1.2.1. c3p0 connection pool

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the `lib/` directory. Hibernate uses its `org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider` for connection pooling if you set the hibernate.c3p0.* properties. properties.

Important configuration properties for the c3p0 connection pool

- hibernate.c3p0.min_size
- hibernate.c3p0.max_size
- hibernate.c3p0.timeout
- hibernate.c3p0.max_statements

1.2.2. Proxool connection pool

Proxool is another open source JDBC connection pool distributed along with Hibernate in the `lib/` directory. Hibernate uses its `org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider` for connection pooling if you set the hibernate.proxool.* properties. Unlike c3p0, proxool requires some additional configuration parameters, as described by the Proxool documentation available at http://proxool.sourceforge.net/configure.html.

Table 1.1. Important configuration properties for the Proxool connection pool

Property	Description
hibernate.proxool.xml	Configure Proxool provider using an XML file (.xml is appended automatically)
hibernate.proxool.properties	Configure the Proxool provider using a properties file (.properties is appended automatically)
hibernate.proxool.existing_pool	Whether to configure the Proxool provider from an existing pool
hibernate.proxool.pool_alias	Proxool pool alias to use. Required.

1.2.3. Obtaining connections from an application server, using JNDI

To use Hibernate inside an application server, configure Hibernate to obtain connections from an application server `javax.sql.Datasource` registered in JNDI, by setting at least one of the following properties:

Important Hibernate properties for JNDI datasources

- hibernate.connection.datasource (required)
- hibernate.jndi.url
- hibernate.jndi.class
- hibernate.connection.username
- hibernate.connection.password

JDBC connections obtained from a JNDI datasource automatically participate in the container-managed transactions of the application server.

1.2.4. Other connection-specific configuration

You can pass arbitrary connection properties by prepending `hibernate.connection` to the connection property name. For example, specify a charSet connection property as hibernate.connection.charSet.

You can define your own plugin strategy for obtaining JDBC connections by implementing the interface `org.hibernate.service.jdbc.connections.spi.ConnectionProvider` and specifying your custom implementation with the hibernate.connection.provider_class property.

1.2.5. Optional configuration properties

In addition to the properties mentioned in the previous sections, Hibernate includes many other optional properties. See ??? for a more complete list.

1.3. Dialects

Although SQL is relatively standardized, each database vendor uses a subset of supported syntax. This is referred to as a *dialect*. Hibernate handles variations across these dialects through its `org.hibernate.dialect.Dialect` class and the various subclasses for each vendor dialect.

Table 1.2. Supported database dialects

Database	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Firebird	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Ingres	org.hibernate.dialect.IngresDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect

Database	Dialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle 8i	org.hibernate.dialect.Oracle8iDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL 8.1	org.hibernate.dialect.PostgreSQL81Dialect
PostgreSQL 8.2 and later	org.hibernate.dialect.PostgreSQL82Dialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase ASE 15.5	org.hibernate.dialect.SybaseASE15Dialect
Sybase ASE 15.7	org.hibernate.dialect.SybaseASE157Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

1.3.1. Specifying the Dialect to use

The developer may manually specify the Dialect to use by setting the hibernate.dialect configuration property to the name of a specific `org.hibernate.dialect.Dialect` class to use.

1.3.2. Dialect resolution

Assuming a `org.hibernate.service.jdbc.connections.spi.ConnectionProvider` has been set up, Hibernate will attempt to automatically determine the Dialect to use based on the `java.sql.DatabaseMetaData` reported by a `java.sql.Connection` obtained from that `org.hibernate.service.jdbc.connections.spi.ConnectionProvider`.

This functionality is provided by a series of `org.hibernate.service.jdbc.dialect.spi.DialectResolver` instances registered with Hibernate internally. Hibernate comes with a standard set of recognitions. If your application requires extra Dialect resolution capabilities, it would simply register a custom implementation of `org.hibernate.service.jdbc.dialect.spi.DialectResolver` as follows:

Registered `org.hibernate.service.jdbc.dialect.spi.DialectResolver` are *prepended* to an internal list of resolvers, so they take precedence before any already registered resolvers including the standard one.

1.4. Automatic schema generation with SchemaExport

SchemaExport is a Hibernate utility which generates DDL from your mapping files. The generated schema includes referential integrity constraints, primary and foreign keys, for entity and collection tables. It also creates tables and sequences for mapped identifier generators.

Note

You must specify a SQL Dialect via the hibernate.dialect property when using this tool, because DDL is highly vendor-specific. See Section 1.3, “Dialects” for information.

Before Hibernate can generate your schema, you must customize your mapping files.

1.4.1. Customizing the mapping files

Hibernate provides several elements and attributes to customize your mapping files. They are listed in Table 1.3, “Elements and attributes provided for customizing mapping files”, and a logical order of customization is presented in Procedure 1.1, “Customizing the schema”.

Table 1.3. Elements and attributes provided for customizing mapping files

Name	Type of value	Description
length	number	Column length

Name	Type of value	Description
precision	number	Decimal precision of column
scale	number	Decimal scale of column
not-null	`true` or `false`	Whether a column is allowed to hold null values
unique	`true` or `false`	Whether values in the column must be unique
index	string	The name of a multi-column index
unique-key	string	The name of a multi-column unique constraint
foreign-key	string	The name of the foreign key constraint generated for an association. This applies to <one-to-one>, <many-to-one>, <key>, and <many-to-many> mapping elements. `inverse="true"` sides are skipped by SchemaExport.
sql-type	string	Overrides the default column type. This applies to the <column> element only.
default	string	Default value for the column
check	string	An SQL check constraint on either a column or atable

Procedure 1.1. Customizing the schema

1. **Set the length, precision, and scale of mapping elements.**

   Many Hibernate mapping elements define optional attributes named `length`, `precision`, and `scale`.

   ```
   <property name="zip" length="5"/>

   <property name="balance" precision="12" scale="2"/>
   ```

2. **Set the `not-null`, `UNIQUE`, `unique-key` attributes.**

   The `not-null` and `UNIQUE` attributes generate constraints on table columns.

   The unique-key attribute groups columns in a single, unique key constraint. Currently, the specified value of the unique-key attribute does not name the constraint in the generated DDL. It only groups the columns in the mapping file.

   ```
   <many-to-one name="bar" column="barId" not-null="true"/>

   <element column="serialNumber" type="long" not-null="true" unique="true"/>


   <many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>

   <property name="employeeId" unique-key="OrgEmployee"/>
   ```

3. **Set the `index` and `foreign-key` attributes.**

   The `index` attribute specifies the name of an index for Hibernate to create using the mapped column or columns. You can group multiple columns into the same index by assigning them the same index name.

   A foreign-key attribute overrides the name of any generated foreign key constraint.

   ```
   <many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
   ```

4. **Set child `<column>` elements.**

   Many mapping elements accept one or more child <column> elements. This is particularly useful for mapping types involving multiple columns.

```
<property name="name" type="my.customtypes.Name"/>

    <column name="last" not-null="true" index="bar_idx" length="30"/>

    <column name="first" not-null="true" index="bar_idx" length="20"/>

    <column name="initial"/>

</property>
```

5. **Set the `default` attribute.**

   The `default` attribute represents a default value for a column. Assign the same value to the mapped property before saving a new instance of the mapped class.

```
<property name="credits" type="integer" insert="false">

    <column name="credits" default="10"/>

</property>

<version name="version" type="integer" insert="false">

    <column name="version" default="0"/>

</property>
```

6. **Set the `sql-type` attribure.**

   Use the `sql-type` attribute to override the default mapping of a Hibernate type to SQL datatype.

```
<property name="balance" type="float">
```

```
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

7. **Set the `check` attribute.**

   use the `check` attribute to specify a *check* constraint.

```
<property name="foo" type="integer">
  <column name="foo" check="foo > 10"/>
</property>
<class name="Foo" table="foos" check="bar < 100.0">
  ...
  <property name="bar" type="float"/>
</class>
```

8. **Add <comment> elements to your schema.**

   Use the <comment> element to specify comments for the generated schema.

```
<class name="Customer" table="CurCust">
  <comment>Current customers only</comment>
  ...
```

```
</class>
```

1.4.2. Running the SchemaExport tool

The SchemaExport tool writes a DDL script to standard output, executes the DDL statements, or both.

Example 1.7. SchemaExport syntax

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaExport options mapping_files
```

Table 1.4. SchemaExport Options

Option	Description
--quiet	do not output the script to standard output
--drop	only drop the tables
--create	only create the tables
--text	do not export to the database
--output=*my_schema.ddl*	output the ddl script to a file
--naming=*eg.MyNamingStrategy*	select a NamingStrategy
--config=*hibernate.cfg.xml*	read Hibernate configuration from an XML file
--properties=*hibernate.properties*	read database properties from a file
--format	format the generated SQL nicely in the script
--delimiter=*;*	set an end-of-line delimiter for the script

Example 1.8. Embedding SchemaExport into your application

```
Configuration cfg = ....;

new SchemaExport(cfg).create(false, true);
```

Chapter 2. Transactions and concurrency control

Table of Contents

2.1. Defining Transaction

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.

Note

This documentation largely treats the physical and logic notions of transaction as one-in-the-same.

2.2. Physical Transactions

Hibernate uses the JDBC API for persistence. In the world of Java there are 2 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.

The first concept in understanding Hibernate transaction support is the `org.hibernate.engine.transaction.spi.TransactionFactory` interface which serves 2 main functions:

- It allows Hibernate to understand the transaction semantics of the environment. Are we operating in a JTA environment? Is a physical transaction already currently active? etc.
- It acts as a factory for `org.hibernate.Transaction` instances which are used to allow applications to manage and check the state of transactions. `org.hibernate.Transaction` is Hibernate's notion of a logical transaction. JPA has a similar notion in the `javax.persistence.EntityTransaction` interface.

Note

`javax.persistence.EntityTransaction` is only available when using resource-local transactions. Hibernate allows access to `org.hibernate.Transaction` regardless of environment.

`org.hibernate.engine.transaction.spi.TransactionFactory` is a standard Hibernate service. See Section 7.5.16, “`org.hibernate.engine.transaction.spi.TransactionFactory`” for details.

2.2.1. Physical Transactions - JDBC

JDBC-based transaction management leverages the JDBC defined methods `java.sql.Connection.commit()` and `java.sql.Connection.rollback()` (JDBC does not define an explicit method of beginning a transaction). In Hibernate, this approach is represented by the `org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory` class.

2.2.2. Physical Transactions - JTA

JTA-based transaction approach which leverages the `javax.transaction.UserTransaction` interface as obtained from `org.hibernate.service.jta.platform.spi.JtaPlatform` API. This approach is represented by the `org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory` class.

See Section 7.5.9, "`org.hibernate.service.jta.platform.spi.JtaPlatform`" for information on integration with the underlying JTA system.

2.2.3. Physical Transactions - CMT

Another JTA-based transaction approach which leverages the JTA `javax.transaction.TransactionManager` interface as obtained from `org.hibernate.service.jta.platform.spi.JtaPlatform` API. This approach is represented by the `org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory` class. In an actual JEE CMT environment, access to the `javax.transaction.UserTransaction` is restricted.

Note

The term CMT is potentially misleading here. The important point simply being that the physical JTA transactions are being managed by something other than the Hibernate transaction API.

See Section 7.5.9, "`org.hibernate.service.jta.platform.spi.JtaPlatform`" for information on integration with the underlying JTA system.

2.2.4. Physical Transactions - Custom

Its is also possible to plug in a custom transaction approach by implementing the `org.hibernate.engine.transaction.spi.TransactionFactory` contract. The default service initiator has built-in support for understanding custom transaction approaches via the `hibernate.transaction.factory_class` which can name either:

- The instance of `org.hibernate.engine.transaction.spi.TransactionFactory` to use.
- The name of a class implementing `org.hibernate.engine.transaction.spi.TransactionFactory` to use. The expectation is that the implementation class have a no-argument constructor.

2.2.5. Physical Transactions - Legacy

During development of 4.0, most of these classes named here were moved to new packages. To help facilitate upgrading, Hibernate will also recognize the legacy names here for a short period of time.

- `org.hibernate.transaction.JDBCTransactionFactory` is mapped to `org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory`
- `org.hibernate.transaction.JTATransactionFactory` is mapped to `org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory`
- `org.hibernate.transaction.CMTTransactionFactory` is mapped to `org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory`

2.3. Hibernate Transaction Usage

Hibernate uses JDBC connections and JTA resources directly, without adding any additional locking behavior. It is important for you to become familiar with the JDBC, ANSI SQL, and transaction isolation specifics of your database management system.

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 `org.hibernate.Session` acts as a transaction-scoped cache providing repeatable reads for lookup by identifier and queries that result in loading entities.

Important

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`.

2.4. Transactional patterns (and anti-patterns)

2.4.1. 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.

> **Note**
>
> 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.

2.4.2. 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 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`'s `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

Important

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.

2.4.3. 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:

Procedure 2.1. An example of a long-running conversation

1. 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.
2. 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 if a concurrent modification occurred during user think time. Check for this at the end of the conversation.
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.

2.4.4. Session-per-application

Discussion coming soon..

2.5. Object identity

An application can concurrently access the same persistent state (database row) in two different Sessions. However, an instance of a persistent class is never shared between two `Session` instances. Two different notions of identity exist and come into play here: Database identity and JVM identity.

Example 2.1. Database identity

```
foo.getId().equals( bar.getId() )
```

Example 2.2. JVM identity

```
foo==bar
```

For objects attached to a particular `Session`, the two notions are equivalent, and JVM identity for database identity is guaranteed by Hibernate. The application might concurrently access a business object with the same identity in two different sessions, the two instances are actually different, in terms of JVM identity. Conflicts are resolved using an optimistic approach and automatic versioning at flush/commit time.

This approach places responsibility for concurrency on Hibernate and the database. It also provides the best scalability, since expensive locking is not needed to guarantee identity in single-threaded units of work. The application does not need to synchronize on any business object, as long as it maintains a single thread per anti-patterns. While not recommended, within a `Session` the application could safely use the `==` operator to compare objects.

However, an application that uses the `==` operator outside of a `Session` may introduce problems.. If you put two detached instances into the same `Set`, they might use the same database identity, which means they represent the same row in the database. They would not be guaranteed to have the same JVM identity if they are in a detached state. Override the `equals` and `hashCode` methods in persistent classes, so that they have their own notion of object equality. Never use the database identifier to implement equality. Instead, use a business key that is a combination of unique, typically immutable, attributes. The database identifier changes if a transient object is made persistent. If the transient instance, together with detached instances, is held in a `Set`, changing the hash-code breaks the contract of the `Set`. Attributes for business keys can be less stable than database primary keys. You only need to guarantee stability as long as the objects are in the same `Set`.This is not a Hibernate issue, but relates to Java's implementation of object identity and equality.

2.6. Common issues

Both the *session-per-user-session* and *session-per-application* anti-patterns are susceptible to the following issues. Some of the issues might also arise within the recommended patterns, so ensure that you understand the implications before making a design decision:

- A `Session` is not thread-safe. Things that work concurrently, like HTTP requests, session beans, or Swing workers, will cause race conditions if a `Session` instance is shared. If you keep your Hibernate `Session` in your `javax.servlet.http.HttpSession` (this is discussed later in the chapter), you should consider synchronizing access to your `HttpSession`; otherwise, a user that clicks reload fast enough can use the same `Session` in two concurrently running threads.
- An exception thrown by Hibernate means you have to rollback your database transaction and close the `Session` immediately (this is discussed in more detail later in the chapter). 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 changes 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 an alternate means of dealing with large amounts of data such as a Stored Procedure. Java is simply not the right tool for these kind of operations. Some solutions are shown in Chapter 4, *Batch Processing*. Keeping a `Session` open for the duration of a user session also means a higher probability of stale data.

Chapter 3. Persistence Contexts

Table of Contents

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.

Entity states

- `new`, or `transient` - the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- `managed`, or `persistent` - the entity has an associated identifier and is associated with a persistence context.
- `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.

In Hibernate native APIs, the persistence context is defined as the `org.hibernate.Session`. In JPA, the persistence context is defined by `javax.persistence.EntityManager`. Much of the `org.hibernate.Session` and `javax.persistence.EntityManager` methods deal with moving entities between these states.

3.1. 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 3.1. Example of making an entity persistent

```
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
session.save( fritz );
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
entityManager.persist( fritz );
```

`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 method on `org.hibernate.Session` to which the Hibernate `javax.persistence.EntityManager` implementation delegates.

If the `DomesticCat` entity type has a generated identifier, the value is associated to the instance when the `save` or `persist` is called. If the identifier is not automatically generated, the application-assigned (usually natural) key value has to be set on the instance before `save` or `persist` is called.

3.2. Deleting entities

Entities can also be deleted.

Example 3.2. Example of deleting an entity

```
session.delete( fritz );
entityManager.remove( fritz );
```

It is important to note that 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.

3.3. 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 3.3. Example of obtaining an entity reference without initializing its data

```
Book book = new Book();
book.setAuthor( session.byId( Author.class ).getReference( authorId ) );
Book book = new Book();
book.setAuthor( entityManager.getReference( Author.class, authorId ) );
```

The above works on the assumption that the entity is defined to allow lazy loading, generally through use of runtime proxies. For more information see ???. In both cases an exception will be thrown later if the given entity does not refer to actual database state if and when the application attempts to use the returned proxy in any way that requires access to its data.

3.4. Obtain an entity with its data initialized

It is also quite common to want to obtain an entity along with with its data, for display for example.

Example 3.4. Example of obtaining an entity reference with its data initialized

```
session.byId( Author.class ).load( authorId );
entityManager.find( Author.class, authorId );
```

In both cases null is returned if no matching database row was found.

3.5. Obtain an entity by natural-id

In addition to allowing to load by identifier, Hibernate allows applications to load by declared natural identifier.

Example 3.5. Example of simple natural-id access

```
@Entity
public class User {
        @Id
        @GeneratedValue
        Long id;

        @NaturalId
        String userName;

        ...
}

// use getReference() to create associations...
Resource aResource = (Resource) session.byId( Resource.class ).getReference( 123 );
User aUser = (User) session.bySimpleNaturalId( User.class ).getReference( "steve" );
aResource.assignTo( user );


// use load() to pull initialzed data
return session.bySimpleNaturalId( User.class ).load( "steve" );
```

Example 3.6. Example of natural-id access

```
import java.lang.String;
```

```
@Entity
public class User {
        @Id
        @GeneratedValue
        Long id;

        @NaturalId
        String system;

        @NaturalId
        String userName;

        ...
}

// use getReference() to create associations...
Resource aResource = (Resource) session.byId( Resource.class ).getReference( 123 );
User aUser = (User) session.byNaturalId( User.class )
                .using( "system", "prod" )
                .using( "userName", "steve" )
                .getReference();
aResource.assignTo( user );


// use load() to pull initialzed data
return session.byNaturalId( User.class )
                .using( "system", "prod" )
                .using( "userName", "steve" )
                .load();
```

Just like we saw above, access entity data by natural id allows both the `load` and `getReference` forms, with the same semantics.

Accessing persistent data by identifier and by natural-id is consistent in the Hibernate API. Each defines the same 2 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.

In addition to those 2 methods, each also defines the method `with` accepting a `org.hibernate.LockOptions` argument. Locking is discussed in a separate chapter.

3.6. Refresh entity state

You can reload an entity instance and it's collections at any time.

Example 3.7. Example of refreshing entity state

```
Cat cat = session.get( Cat.class, catId );
...
session.refresh( cat );
Cat cat = entityManager.find( Cat.class, catId );
...
entityManager.refresh( cat );
```

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. Note that only the entity instance and its 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 ??? for information.

3.7. 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 3.8. Example of modifying managed state

```
Cat cat = session.get( Cat.class, catId );
cat.setName( "Garfield" );
session.flush(); // generally this is not explicitly needed
Cat cat = entityManager.find( Cat.class, catId );
cat.setName( "Garfield" );
entityManager.flush(); // generally this is not explicitly needed
```

3.8. 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.

3.8.1. 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.

Important

JPA does not provide for this model. This is only available through Hibernate `org.hibernate.Session`.

Example 3.9. Example of reattaching a detached entity

```
session.saveOrUpdate( someDetachedCat );
```

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. Unless the entity is mapped with `select-before-update`, in which case 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.

3.8.2. 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 instance that is in managed state.

Example 3.10. Visualizing merge

```
Object detached = ...;
Object managed = entityManager.find( detached.getClass(), detached.getId() );
managed.setXyz( detached.getXyz() );
...
return managed;
```

That is not exactly what happens, but its a good visualization.

Example 3.11. Example of merging a detached entity

```
Cat theManagedInstance = session.merge( someDetachedCat );
Cat theManagedInstance = entityManager.merge( someDetachedCat );
```

3.9. Checking persistent state

An application can verify the state of entities and collections in relation to the persistence context.

Example 3.12. Examples of verifying managed state

```
assert session.contains( cat );
assert entityManager.contains( cat );
```

Example 3.13. Examples of verifying laziness

```
if ( Hibernate.isInitialized( customer.getAddress() ) {
    //display address if loaded
}
if ( Hibernate.isInitialized( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (Hibernate.isPropertyInitialized( customer, "detailedBio" ) ) {
    //display property detailedBio if loaded
}
javax.persistence.PersistenceUnitUtil jpaUtil = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();
if ( jpaUtil.isLoaded( customer.getAddress() ) {
    //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (jpaUtil.isLoaded( customer, "detailedBio" ) ) {
    //display property detailedBio if loaded
}
```

In JPA there is an alternative means to check laziness using the following `javax.persistence.PersistenceUtil` pattern. However, the `javax.persistence.PersistenceUnitUtil` is recommended where ever possible

Example 3.14. Alternative JPA means to verify laziness

```
javax.persistence.PersistenceUtil jpaUtil = javax.persistence.Persistence.getPersistenceUtil();
if ( jpaUtil.isLoaded( customer.getAddress() ) {
    //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (jpaUtil.isLoaded(customer, "detailedBio") ) {
    //display property detailedBio if loaded
}
```

3.10. Accessing Hibernate APIs from JPA

JPA defines an incredibly useful method to allow applications access to the APIs of the underlying provider.

Example 3.15. Usage of EntityManager.unwrap

```
Session session = entityManager.unwrap( Session.class );
SessionImplementor sessionImplementor = entityManager.unwrap( SessionImplementor.class );
```

Chapter 4. Batch Processing

Table of Contents

The following example shows an antipattern for batch inserts.

Example 4.1. Naive way to insert 100000 lines with Hibernate

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

    Customer customer = new Customer(.....);

    session.save(customer);

}

tx.commit();

session.close();
```

This fails with exception `OutOfMemoryException` after around 50000 rows on most systems. The reason is that Hibernate caches all the newly inserted Customer instances in the session-level cache. There are several ways to avoid this problem.

Before batch processing, enable JDBC batching. To enable JDBC batching, set the property hibernate.jdbc.batch_size to an integer between 10 and 50.

Note

Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.

If the above approach is not appropriate, you can disable the second-level cache, by setting hibernate.cache.use_second_level_cache to `false`.

4.1. 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 4.2. Flushing and clearing the `Session`

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


for ( int i=0; i<100000; i++ ) {

    Customer customer = new Customer(.....);

    session.save(customer);

    if ( i % 20 == 0 ) { //20, same as the JDBC batch size

        //flush a batch of inserts and release memory:

        session.flush();

        session.clear();

    }

}


tx.commit();

session.close();
```

4.2. Batch updates

When you retriev 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 4.3. Using `scroll()`

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


ScrollableResults customers = session.getNamedQuery("GetCustomers")

    .setCacheMode(CacheMode.IGNORE)

    .scroll(ScrollMode.FORWARD_ONLY);

int count=0;

while ( customers.next() ) {

    Customer customer = (Customer) customers.get(0);

    customer.updateStuff(...);

    if ( ++count % 20 == 0 ) {

        //flush a batch of updates and release memory:

        session.flush();

        session.clear();

    }

}
```

```
tx.commit();

session.close();
```

4.3. 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:

Features and behaviors not provided by `StatelessSession`

- 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.
- 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 4.4. Using a `StatelessSession`

```
StatelessSession session = sessionFactory.openStatelessSession();

Transaction tx = session.beginTransaction();
```

```
ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

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.

4.4. Hibernate Query Language for DML

DML, or *Data Markup 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)*.

4.4.1. HQL for UPDATE and DELETE

Example 4.5. Psuedo-syntax for UPDATE and DELETE statements using HQL

```
          ( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?
```

The `?` suffix indications an optional parameter. The `FROM` and `WHERE` clauses are each optional.

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 4.6. Executing an HQL UPDATE, using the `Query.executeUpdate()` method

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";

// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";

int updatedEntities = s.createQuery( hqlUpdate )

        .setString( "newName", newName )

        .setString( "oldName", oldName )

        .executeUpdate();
```

```
tx.commit();

session.close();
```

In keeping with the EJB3 specification, HQL UPDATE statements, by default, do not effect the version or the timestamp property values for the affected entities. 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 4.7. Updating the version of timestamp

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";

int updatedEntities = s.createQuery( hqlUpdate )

        .setString( "newName", newName )

        .setString( "oldName", oldName )

        .executeUpdate();

tx.commit();

session.close();
```

Note

If you use the `VERSIONED` statement, you cannot use custom version types, which use class `org.hibernate.usertype.UserVersionType`.

Example 4.8. A HQL `DELETE` statement

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


String hqlDelete = "delete Customer c where c.name = :oldName";

// or String hqlDelete = "delete Customer where name = :oldName";

int deletedEntities = s.createQuery( hqlDelete )

        .setString( "oldName", oldName )

        .executeUpdate();

tx.commit();

session.close();
```

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 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.

4.4.2. HQL syntax for INSERT

Example 4.9. Pseudo-syntax for INSERT statements

```
          INSERT INTO EntityName properties_list select_statement
```

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 4.10. HQL INSERT statement

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";

int createdEntities = s.createQuery( hqlInsert )

        .executeUpdate();

tx.commit();

session.close();
```

4.4.3. More information on HQL

This section is only a brief overview of HQL. For more information, see ???.

Chapter 5. Locking

Table of Contents

Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Your locking strategy can be either *optimistic* or *pessimistic*.

Locking strategies

Optimistic

> Optimistic locking ssumes 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[1].

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.

5.1. 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.

Version number
Timestamp

Note

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.

5.1.1. Dedicated version number

The version number mechanism for optimistic locking is provided through a `@Version` annotation.

Example 5.1. The @Version annotation

```
@Entity

public class Flight implements Serializable {

...

    @Version
    @Column(name="OPTLOCK")

    public Integer getVersion() { ... }

}
```

Here, the version property is mapped to the `OPTLOCK` column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would 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_INCREMENTcheck in the Hibernate Entity Manager reference documentation.

Database-generated version numbers

If the version number is generated by the database, such as a trigger, use the annotation `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

Example 5.2. Declaring a version property in `hbm.xml`

```
<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`.

5.1.2. 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.

Example 5.3. Using timestamps for optimistic locking

```
@Entity

public class Flight implements Serializable {
```

```
...
    @Version
    public Date getLastUpdate() { ... }
}
```

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.

Example 5.4. The timestamp element in `hbm.xml`

```
<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 namel
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`.

5.2. 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.

Note

Hibernate always uses the locking mechanism of the database, and never lock objects in memory.

5.2.1. The `LockMode` class

The `LockMode` class defines the different lock levels that Hibernate can acquire.

LockMode.WRITE	acquired automatically when Hibernate updates or inserts a row.
LockMode.UPGRADE	acquired upon explicit user request using `SELECT ... FOR UPDATE` on databases which support that syntax.
LockMode.UPGRADE_NOWAIT	acquired upon explicit user request using a `SELECT ... FOR UPDATE NOWAIT` in Oracle.

LockMode.READ	acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request.
LockMode.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.

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` or `UPGRADE_NOWAIT`, 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`, or `UPGRADE_NOWAIT`. In the case of `UPGRADE` or `UPGRADE_NOWAIT`, `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.

---

[1] http://en.wikipedia.org/wiki/Optimistic_locking

Chapter 6. Caching

Table of Contents

6.1. The query cache

If you have queries that run over and over, with the same parameters, query caching provides performance gains.

Caching introduces overhead in the area of transactional processing. For example, if you cache results of a query against an object, Hibernate needs to keep track of whether any changes have been committed against the object, and invalidate the cache accordingly. In addition, the benefit from caching query results is limited, and highly dependent on the usage patterns of your application. For these reasons, Hibernate disables the query cache by default.

Procedure 6.1. Enabling the query cache

1. **Set the hibernate.cache.use_query_cache property to `true`.**

   This setting creates two new cache regions:

   - `org.hibernate.cache.internal.StandardQueryCache` holds the cached query results.
   - `org.hibernate.cache.spi.UpdateTimestampsCache` holds timestamps of the most recent updates to queryable tables. These timestamps validate results served from the query cache.

2. **Adjust the cache timeout of the underlying cache region**

   If you configure your underlying cache implementation to use expiry or timeouts, set the cache timeout of the underlying cache region for the `UpdateTimestampsCache` to a higher value than the timeouts of any of the query caches. It is possible, and recommended, to set the UpdateTimestampsCache region never to expire. To be specific, a LRU (Least Recently Used) cache expiry policy is never appropriate.

3. **Enable results caching for specific queries**

   Since most queries do not benefit from caching of their results, you need to enable caching for individual queries, e ven after enabling query caching overall. To enable results caching for a particular query, call `org.hibernate.Query.setCacheable(true)`. This call allows the query to look for existing cache results or add its results to the cache when it is executed.

The query cache does not cache the state of the actual entities in the cache. It caches identifier values and results of value type. Therefore, always use the query cache in conjunction with the second-level cache for those entities which should be cached as part of a query result cache.

6.1.1. Query cache regions

For fine-grained control over query cache expiration policies, specify a named cache region for a particular query by calling `Query.setCacheRegion()`.

Example 6.1. Method `setCacheRegion`

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")

        .setEntity("blogger", blogger)

        .setMaxResults(15)

        .setCacheable(true)

        .setCacheRegion("frontpages")

        .list();
```

To force the query cache to refresh one of its regions and disregard any cached results in the region, call `org.hibernate.Query.setCacheMode(CacheMode.REFRESH)`. In conjunction with the region defined for the given query, Hibernate selectively refreshes the results cached in that particular region. This is much more efficient than bulk eviction of the region via `org.hibernate.SessionFactory.evictQueries()`.

6.2. Second-level cache providers

Hibernate is compatible with several second-level cache providers. None of the providers support all of Hibernate's possible caching strategies. Section 6.2.3, “Second-level cache providers for Hibernate” lists the providers, along with their interfaces and supported caching strategies. For definitions of caching strategies, see Section 6.2.2, “Caching strategies”.

6.2.1. Configuring your cache providers

You can configure your cache providers using either annotations or mapping files.

Entities. By default, entities are not part of the second-level cache, and their use is not recommended. If you absolutely must use entities, set the `shared-cache-mode` element in `persistence.xml`, or use property javax.persistence.sharedCache.mode in your configuration. Use one of the values in Table 6.1, “Possible values for Shared Cache Mode”.

Table 6.1. Possible values for Shared Cache Mode

Value	Description
ENABLE_SELECTIVE	Entities are not cached unless you explicitly mark them as cachable. This is the default and recommended value.
DISABLE_SELECTIVE	Entities are cached unless you explicitly mark them as not cacheable.
ALL	All entities are always cached even if you mark them as not cacheable.
NONE	No entities are cached even if you mark them as cacheable. This option basically disables second-level caching.

Set the global default cache concurrency strategy The cache concurrency strategy with the hibernate.cache.default_cache_concurrency_strategy configuration property. See Section 6.2.2, “Caching strategies” for possible values.

Note

When possible, define the cache concurrency strategy per entity rather than globally. Use the `@org.hibernate.annotations.Cache` annotation.

Example 6.2. Configuring cache providers using annotations

```
@Entity

@Cacheable

@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)

public class Forest { ... }
```

You can cache the content of a collection or the identifiers, if the collection contains other entities. Use the `@Cache` annotation on the Collection property.

`@Cache` can take several attributes.

Attributes of `@Cache` annotation

usage

The given cache concurrency strategy, which may be:

- `NONE`
- `READ_ONLY`
- `NONSTRICT_READ_WRITE`
- `READ_WRITE`
- `TRANSACTIONAL`

region

The cache region. This attribute is optional, and defaults to the fully-qualified class name of the class, or the qually-qualified role name of the collection.

include

Whether or not to include all properties.. Optional, and can take one of two possible values.

- A value of `all` includes all properties. This is the default.
- A value of `non-lazy` only includes non-lazy properties.

Example 6.3. Configuring cache providers using mapping files

```
<cache

    usage="transactional"

    region="RegionName"

    include="all"

/>
```

Just as in the Example 6.2, “Configuring cache providers using annotations”, you can provide attributes in the mapping file. There are some specific differences in the syntax for the attributes in a mapping file.

usage

The caching strategy. This attribute is required, and can be any of the following values.

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

region

The name of the second-level cache region. This optional attribute defaults to the class or collection role name.

include

Whether properties of the entity mapped with `lazy=true` can be cached when attribute-level lazy fetching is enabled. Defaults to `all` and can also be `non-lazy`.

Instead of `<cache>`, you can use `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

6.2.2. Caching strategies

read-only

A read-only cache is good for data that needs to be read often but not modified. It is simple, performs well, and is safe to use in a clustered environment.

nonstrict read-write

Some applications only rarely need to modify data. This is the case if two transactions are unlikely to try to update the same item simultaneously. In this case, you do not need strict transaction isolation, and a nonstrict-read-write cache might be appropriate. If the cache is used in a JTA environment, you must specify `hibernate.transaction.manager_lookup_class`. In other environments, ensore that the transaction is complete before you call `Session.close()` or `Session.disconnect()`.

read-write

A read-write cache is appropriate for an application which needs to update data regularly. Do not use a read-write strategy if you need serializable transaction isolation. In a JTA environment, specify a strategy for obtaining the JTA TransactionManager by setting the property hibernate.transaction.manager_lookup_class. In non-JTA environments, be sure the transaction is complete before you call `Session.close()` or `Session.disconnect()`.

Note

To use the read-write strategy in a clustered environment, the underlying cache implementation must support locking. The build-in cache providers do not support locking.

transactional

The transactional cache strategy provides support for transactional cache providers such as JBoss TreeCache. You can only use such a cache in a JTA environment, and you must first specify `hibernate.transaction.manager_lookup_class`.

6.2.3. Second-level cache providers for Hibernate

Cache	Interface	Supported strategies		
HashTable (testing only)		• read-only • nontrict read-write • read-write		
EHCache		• read-only • nontrict read-write • read-write		
OSCache		• read-only • nontrict read-write • read-write		
SwarmCache		• read-only • nontrict read-write		
JBoss Cache 1.x		• read-only • transactional		
JBoss Cache 2.x		• read-only • transactional		

6.3. Managing the cache

6.3.1. Moving items into and out of the cache

Actions that add an item to internal cache of the Session

Saving or updating an item

- `save()`
- `update()`
- `saveOrUpdate()`

Retrieving an item

- `load()`
- `get()`
- `list()`
- `iterate()`
- `scroll()`

Syncing or removing a cached item. The state of an object is synchronized with the database when you call method `flush()`. To avoid this synchronization, you can remove the object and all collections from the first-level cache with the `evict()` method. To remove all items from the Session cache, use method `Session.clear()`.

Example 6.4. Evicting an item from the first-level cache

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set

while ( cats.next() ) {

    Cat cat = (Cat) cats.get(0);

    doSomethingWithACat(cat);

    sess.evict(cat);

}
```

Determining whether an item belongs to the Session cache. The Session provides a `contains()` method to determine if an instance belongs to the session cache.

Example 6.5. Second-level cache eviction

You can evict the cached state of an instance, entire class, collection instance or entire collection role, using methods of `SessionFactory`.

```
sessionFactory.getCache().containsEntity(Cat.class, catId); // is this particular Cat currently in the cache

sessionFactory.getCache().evictEntity(Cat.class, catId); // evict a particular Cat

sessionFactory.getCache().evictEntityRegion(Cat.class);  // evict all Cats

sessionFactory.getCache().evictEntityRegions();  // evict all entity data

sessionFactory.getCache().containsCollection("Cat.kittens", catId); // is this particular collection currently in the cache

sessionFactory.getCache().evictCollection("Cat.kittens", catId); // evict a particular collection of kittens

sessionFactory.getCache().evictCollectionRegion("Cat.kittens"); // evict all kitten collections

sessionFactory.getCache().evictCollectionRegions(); // evict all collection data
```

6.3.1.1. Interactions between a Session and the second-level cache

The CacheMode controls how a particular session interacts with the second-level cache.

CacheMode.NORMAL	reads items from and writes them to the second-level cache.

CacheMode.GET	reads items from the second-level cache, but does not write to the second-level cache except to update data.
CacheMode.PUT	writes items to the second-level cache. It does not read from the second-level cache. It bypasses the effect of hibernate.cache.use_minimal_puts and forces a refresh of the second-level cache for all items read from the database.

6.3.1.2. Browsing the contents of a second-level or query cache region

After enabling statistics, you can browse the contents of a second-level cache or query cache region.

Procedure 6.2. Enabling Statistics

1. Set `hibernate.generate_statistics` to `true`.
2. Optionally, set `hibernate.cache.use_structured_entries` to `true`, to cause Hibernate to store the cache entries in a human-readable format.

Example 6.6. Browsing the second-level cache entries via the Statistics API

```
Map cacheEntries = sessionFactory.getStatistics()

        .getSecondLevelCacheStatistics(regionName)

        .getEntries();
```

Chapter 7. Services

Table of Contents

7.1. What are services?

Services are classes that provide Hibernate with pluggable implementations of various types of functionality. Specifically they are implementations of certain service contract interfaces. The interface is known as the service role; the implementation class is know as the service implementation. Generally speaking, users can plug in alternate implementations of all standard service roles (overriding); they can also define additional services beyond the base set of service roles (extending).

7.2. Service contracts

The basic requirement for a service is to implement the marker interface `org.hibernate.service.Service`. Hibernate uses this internally for some basic type safety.

Optionally, the service can also implement the `org.hibernate.service.spi.Startable` and `org.hibernate.service.spi.Stoppable` interfaces to receive notifications of being started and stopped. Another optional service contract is `org.hibernate.service.spi.Manageable` which marks the service as manageable in JMX provided the JMX integration is enabled.

7.3. Service dependencies

Services are allowed to declare dependencies on other services using either of 2 approaches.

7.3.1. @`org.hibernate.service.spi.InjectService`

Any method on the service implementation class accepting a single parameter and annotated with @`InjectService` is considered requesting injection of another service.

By default the type of the method parameter is expected to be the service role to be injected. If the parameter type is different than the service role, the `serviceRole` attribute of the `InjectService` should be used to explicitly name the role.

By default injected services are considered required, that is the start up will fail if a named dependent service is missing. If the service to be injected is optional, the `required` attribute of the `InjectService` should be declared as `false` (default is `true`).

7.3.2. `org.hibernate.service.spi.ServiceRegistryAwareService`

The second approach is a pull approach where the service implements the optional service interface `org.hibernate.service.spi.ServiceRegistryAwareService` which declares a single `injectServices` method. During startup, Hibernate will inject the `org.hibernate.service.ServiceRegistry` itself into services which implement this interface. The service can then use the `ServiceRegistry` reference to locate any additional services it needs.

7.4. ServiceRegistry

The central service API, aside from the services themselves, is the `org.hibernate.service.ServiceRegistry` interface. The main purpose of a service registry is to hold, manage and provide access to services.

Service registries are hierarchical. Services in one registry can depend on and utilize services in that same registry as well as any parent registries.

Use `org.hibernate.service.ServiceRegistryBuilder` to build a `org.hibernate.service.ServiceRegistry` instance.

7.5. Standard services

7.5.1. `org.hibernate.engine.jdbc.batch.spi.BatchBuilder`

Notes

Defines strategy for how Hibernate manages JDBC statement batching

Initiator

`org.hibernate.engine.jdbc.batch.internal.BatchBuilderInitiator`

Implementations

`org.hibernate.engine.jdbc.batch.internal.BatchBuilderImpl`

7.5.2. `org.hibernate.service.config.spi.ConfigurationService`

Notes

Provides access to the configuration settings, combining those explicitly provided as well as those contributed by any registered `org.hibernate.integrator.spi.Integrator` implementations

Initiator

`org.hibernate.service.config.internal.ConfigurationServiceInitiator`

Implementations

`org.hibernate.service.config.internal.ConfigurationServiceImpl`

7.5.3. `org.hibernate.service.jdbc.connections.spi.ConnectionProvider`

Notes

Defines the means in which Hibernate can obtain and release `java.sql.Connection` instances for its use.

Initiator

`org.hibernate.service.jdbc.connections.internal.ConnectionProviderInitiator`

Implementations

- `org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider` - provides connection pooling based on integration with the C3P0 connection pooling library
- `org.hibernate.service.jdbc.connections.internal.DatasourceConnectionProviderImpl` - provides connection managed delegated to a `javax.sql.DataSource`
- `org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl` - provides rudimentary connection pooling based on simple custom pool. Note intended production use!
- `org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider` - provides connection pooling based on integration with the proxool connection pooling library

- `org.hibernate.service.jdbc.connections.internal.UserSuppliedConnectionProviderImpl` - Provides no connection support. Indicates the user will supply connections to Hibernate directly. Not recommended for use.

7.5.4. `org.hibernate.service.jdbc.dialect.spi.DialectFactory`

Notes

Contract for Hibernate to obtain `org.hibernate.dialect.Dialect` instance to use. This is either explicitly defined by the hibernate.dialect property or determined by the Section 7.5.5, "`org.hibernate.service.jdbc.dialect.spi.DialectResolver`" service which is a delegate to this service.

Initiator

`org.hibernate.service.jdbc.dialect.internal.DialectFactoryInitiator`

Implementations

`org.hibernate.service.jdbc.dialect.internal.DialectFactoryImpl`

7.5.5. `org.hibernate.service.jdbc.dialect.spi.DialectResolver`

Notes

Provides resolution of `org.hibernate.dialect.Dialect` to use based on information extracted from JDBC metadata.

The standard resolver implementation acts as a chain, delegating to a series of individual resolvers. The standard Hibernate resolution behavior is contained in `org.hibernate.service.jdbc.dialect.internal.StandardDialectResolver`. `org.hibernate.service.jdbc.dialect.internal.DialectResolverInitiator` also consults with the hibernate.dialect_resolvers setting for any custom resolvers.

Initiator

`org.hibernate.service.jdbc.dialect.internal.DialectResolverInitiator`

Implementations

`org.hibernate.service.jdbc.dialect.internal.DialectResolverSet`

7.5.6. `org.hibernate.engine.jdbc.spi.JdbcServices`

Notes

Special type of service that aggregates together a number of other services and provides a higher-level set of functionality.

Initiator

`org.hibernate.engine.jdbc.internal.JdbcServicesInitiator`

Implementations

`org.hibernate.engine.jdbc.internal.JdbcServicesImpl`

7.5.7. `org.hibernate.service.jmx.spi.JmxService`

Notes

Provides simplified access to JMX related features needed by Hibernate.

Initiator

`org.hibernate.service.jmx.internal.JmxServiceInitiator`

Implementations

- `org.hibernate.service.jmx.internal.DisabledJmxServiceImpl` - A no-op implementation when JMX functionality is disabled.
- `org.hibernate.service.jmx.internal.JmxServiceImpl` - Standard implementation of JMX handling

7.5.8. `org.hibernate.service.jndi.spi.JndiService`

Notes

Provides simplified access to JNDI related features needed by Hibernate.

Initiator

`org.hibernate.service.jndi.internal.JndiServiceInitiator`

Implementations

`org.hibernate.service.jndi.internal.JndiServiceImpl`

7.5.9. `org.hibernate.service.jta.platform.spi.JtaPlatform`

Notes

Provides an abstraction from the underlying JTA platform when JTA features are used.

Initiator

`org.hibernate.service.jta.platform.internal.JtaPlatformInitiator`

Important

`JtaPlatformInitiator` provides mapping against the legacy, now-deprecated `org.hibernate.transaction.TransactionManagerLookup` names internally for the Hibernate-provided `org.hibernate.transaction.TransactionManagerLookup` implementations.

Implementations

- `org.hibernate.service.jta.platform.internal.BitronixJtaPlatform` - Integration with the Bitronix stand-alone transaction manager.

- `org.hibernate.service.jta.platform.internal.BorlandEnterpriseServerJtaPlatform` - Integration with the transaction manager as deployed within a Borland Enterprise Server
- `org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform` - Integration with the transaction manager as deployed within a JBoss Application Server
- `org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform` - Integration with the JBoss Transactions stand-alone transaction manager
- `org.hibernate.service.jta.platform.internal.JOTMJtaPlatform` - Integration with the JOTM stand-alone transaction manager
- `org.hibernate.service.jta.platform.internal.JOnASJtaPlatform` - Integration with the JOnAS transaction manager.
- `org.hibernate.service.jta.platform.internal.JRun4JtaPlatform` - Integration with the transaction manager as deployed in a JRun 4 application server.
- `org.hibernate.service.jta.platform.internal.NoJtaPlatform` - No-op version when no JTA set up is configured
- `org.hibernate.service.jta.platform.internal.OC4JJtaPlatform` - Integration with transaction manager as deployed in an OC4J (Oracle) application server.
- `org.hibernate.service.jta.platform.internal.OrionJtaPlatform` - Integration with transaction manager as deployed in an Orion application server.
- `org.hibernate.service.jta.platform.internal.ResinJtaPlatform` - Integration with transaction manager as deployed in a Resin application server.
- `org.hibernate.service.jta.platform.internal.SunOneJtaPlatform` - Integration with transaction manager as deployed in a Sun ONE (7 and above) application server.
- `org.hibernate.service.jta.platform.internal.TransactionManagerLookupBridge` - Provides a bridge to legacy (and deprecated) `org.hibernate.transaction.TransactionManagerLookup` implementations
- `org.hibernate.service.jta.platform.internal.WebSphereExtendedJtaPlatform` - Integration with transaction manager as deployed in a WebSphere Application Server (6 and above).
- `org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform` - Integration with transaction manager as deployed in a WebSphere Application Server (4, 5.0 and 5.1).
- `org.hibernate.service.jta.platform.internal.WeblogicJtaPlatform` - Integration with transaction manager as deployed in a Weblogic application server.

7.5.10. `org.hibernate.service.jdbc.connections.spi.MultiTenantConnectionProvider`

Notes

A variation of Section 7.5.3, "`org.hibernate.service.jdbc.connections.spi.ConnectionProvider`" providing access to JDBC connections in multi-tenant environments.

Initiator

N/A

Implementations

Intended that users provide appropriate implementation if needed.

7.5.11. `org.hibernate.persister.spi.PersisterClassResolver`

Notes

Contract for determining the appropriate `org.hibernate.persister.entity.EntityPersister` or `org.hibernate.persister.collection.CollectionPersister` implementation class to use given an entity or collection mapping.

Initiator

`org.hibernate.persister.internal.PersisterClassResolverInitiator`

Implementations

`org.hibernate.persister.internal.StandardPersisterClassResolver`

7.5.12. `org.hibernate.persister.spi.PersisterFactory`

Notes

Factory for creating `org.hibernate.persister.entity.EntityPersister` and `org.hibernate.persister.collection.CollectionPersister` instances.

Initiator

`org.hibernate.persister.internal.PersisterFactoryInitiator`

Implementations

`org.hibernate.persister.internal.PersisterFactoryImpl`

7.5.13. `org.hibernate.cache.spi.RegionFactory`

Notes

Integration point for Hibernate's second level cache support.

Initiator

`org.hibernate.cache.internal.RegionFactoryInitiator`

Implementations

- `org.hibernate.cache.ehcache.EhCacheRegionFactory`
- `org.hibernate.cache.infinispan.InfinispanRegionFactory`
- `org.hibernate.cache.infinispan.JndiInfinispanRegionFactory`
- `org.hibernate.cache.internal.NoCachingRegionFactory`
- `org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory`

7.5.14. `org.hibernate.service.spi.SessionFactoryServiceRegistryFactory`

Notes

Factory for creating `org.hibernate.service.spi.SessionFactoryServiceRegistry` instances which acts as a specialized `org.hibernate.service.ServiceRegistry` for `org.hibernate.SessionFactory` scoped services. See Section 7.7.2, “SessionFactory registry” for more details.

Initiator

`org.hibernate.service.internal.SessionFactoryServiceRegistryFactoryInitiator`

Implementations

`org.hibernate.service.internal.SessionFactoryServiceRegistryFactoryImpl`

7.5.15. `org.hibernate.stat.Statistics`

Notes

Contract for exposing collected statistics. The statistics are collected through the `org.hibernate.stat.spi.StatisticsImplementor` contract.

Initiator

`org.hibernate.stat.internal.StatisticsInitiator`

Defines a hibernate.stats.factory setting to allow configuring the `org.hibernate.stat.spi.StatisticsFactory` to use internally when building the actual `org.hibernate.stat.Statistics` instance.

Implementations

`org.hibernate.stat.internal.ConcurrentStatisticsImpl`

The default `org.hibernate.stat.spi.StatisticsFactory` implementation builds a `org.hibernate.stat.internal.ConcurrentStatisticsImpl` instance.

7.5.16. `org.hibernate.engine.transaction.spi.TransactionFactory`

Notes

Strategy defining how Hibernate's `org.hibernate.Transaction` API maps to the underlying transaction approach.

Initiator

`org.hibernate.stat.internal.StatisticsInitiator`

Defines a hibernate.stats.factory setting to allow configuring the `org.hibernate.stat.spi.StatisticsFactory` to use internally when building the actual `org.hibernate.stat.Statistics` instance.

Implementations

- `org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory` - A JTA-based strategy in which Hibernate is not controlling the transactions. An important distinction here is that interaction with the underlying JTA implementation is done through the `javax.transaction.TransactionManager`
- `org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory` - A non-JTA strategy in which the transactions are managed using the JDBC `java.sql.Connection`
- `org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory` - A JTA-based strategy in which Hibernate *may* be controlling the transactions. An important distinction here is that interaction with the underlying JTA implementation is done through the `javax.transaction.UserTransaction`

7.5.17. `org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractor`

Notes

Contract for extracting statements from `import.sql` scripts.

Initiator

`org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractorInitiator`

Implementations

- `org.hibernate.tool.hbm2ddl.SingleLineSqlCommandExtractor` treats each line as a complete SQL statement. Comment lines shall start with `--`, `//` or `/*` character sequence.
- `org.hibernate.tool.hbm2ddl.MultipleLinesSqlCommandExtractor` supports instructions/comments and quoted strings spread over multiple lines. Each statement must end with semicolon.

7.6. Custom services

Once a `org.hibernate.service.ServiceRegistry` is built it is considered immutable; the services themselves might accept re-configuration, but immutability here means adding/replacing services. So another role provided by the `org.hibernate.service.ServiceRegistryBuilder` is to allow tweaking of the services that will be contained in the `org.hibernate.service.ServiceRegistry` generated from it.

There are 2 means to tell a `org.hibernate.service.ServiceRegistryBuilder` about custom services.

- Implement a `org.hibernate.service.spi.BasicServiceInitiator` class to control on-demand construction of the service class and add it to the `org.hibernate.service.ServiceRegistryBuilder` via its `addInitiator` method.
- Just instantiate the service class and add it to the `org.hibernate.service.ServiceRegistryBuilder` via its `addService` method.

Either approach the adding a service approach or the adding an initiator approach are valid for extending a registry (adding new service roles) and overriding services (replacing service implementations).

7.7. Special service registries

7.7.1. Boot-strap registry

The boot-strap registry holds services that absolutely have to be available for most things to work. The main service here is the Section 7.7.1.1.1, “`org.hibernate.service.classloading.spi.ClassLoaderService`” which is a perfect example. Even resolving configuration files needs access to class loading services (resource look ups). This is the root registry (no parent) in normal use.

Instances of boot-strap registries are built using the `org.hibernate.service.BootstrapServiceRegistryBuilder` class.

Example 7.1. Using BootstrapServiceRegistryBuilder

```
BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()
        // pass in org.hibernate.integrator.spi.Integrator instances which are not
        // auto-discovered (for whatever reason) but which should be included
        .with( anExplicitIntegrator )
        // pass in a class-loader Hibernate should use to load application classes
        .withApplicationClassLoader( anExplicitClassLoaderForApplicationClasses )
        // pass in a class-loader Hibernate should use to load resources
        .withResourceClassLoader( anExplicitClassLoaderForResources )
        // see BootstrapServiceRegistryBuilder for rest of available methods
        ...
        // finally, build the bootstrap registry with all the above options
        .build();
```

7.7.1.1. Bootstrap registry services

7.7.1.1.1. `org.hibernate.service.classloading.spi.ClassLoaderService`

Hibernate needs to interact with ClassLoaders. However, the manner in which Hibernate (or any library) should interact with ClassLoaders varies based on the runtime environment which is hosting the application. Application servers, OSGi containers, and other modular class loading systems impose very specific class-loading requirements. This service is provides Hibernate an abstraction from this environmental complexity. And just as importantly, it does so in a single-swappable-component manner.

In terms of interacting with a ClassLoader, Hibernate needs the following capabilities:

- the ability to locate application classes
- the ability to locate integration classes
- the ability to locate resources (properties files, xml files, etc)
- the ability to load `java.util.ServiceLoader`

Note

Currently, the ability to load application classes and the ability to load integration classes are combined into a single "load class" capability on the service. That may change in a later release.

7.7.1.1.2. `org.hibernate.integrator.spi.IntegratorService`

Applications, add-ons and others all need to integrate with Hibernate which used to require something, usually the application, to coordinate registering the pieces of each integration needed on behalf of each integrator. The intent of this service is to allow those integrators to be discovered and to have them integrate themselves with Hibernate.

This service focuses on the discovery aspect. It leverages the standard Java `java.util.ServiceLoader` capability provided by the `org.hibernate.service.classloading.spi.ClassLoaderService` in order to discover implementations of the `org.hibernate.integrator.spi.Integrator` contract. Integrators would simply define a file named `/META-INF/services/org.hibernate.integrator.spi.Integrator` and make it available on the classpath. `java.util.ServiceLoader` covers the format of this file in detail, but essentially it list classes by FQN that implement the `org.hibernate.integrator.spi.Integrator` one per line.

See Section 7.9, “Integrators”

7.7.2. SessionFactory registry

While it is best practice to treat instances of all the registry types as targeting a given `org.hibernate.SessionFactory`, the instances of services in this group explicitly belong to a single `org.hibernate.SessionFactory`. The difference is a matter of timing in when they need to be initiated. Generally they need access to the `org.hibernate.SessionFactory` to be initiated. This special registry is `org.hibernate.service.spi.SessionFactoryServiceRegistry`

7.7.2.1. `org.hibernate.event.service.spi.EventListenerRegistry`

Notes

Service for managing event listeners.

Initiator

`org.hibernate.event.service.internal.EventListenerServiceInitiator`

Implementations

`org.hibernate.event.service.internal.EventListenerRegistryImpl`

7.8. Using services and registries

Coming soon...

7.9. Integrators

The `org.hibernate.integrator.spi.Integrator` is intended to provide a simple means for allowing developers to hook into the process of building a functioning SessionFactory. The The `org.hibernate.integrator.spi.Integrator` interface defines 2 methods of interest: `integrate` allows us to hook into the building process; `disintegrate` allows us to hook into a SessionFactory shutting down.

Note

There is a 3rd method defined on `org.hibernate.integrator.spi.Integrator`, an overloaded form of `integrate` accepting a `org.hibernate.metamodel.source.MetadataImplementor` instead of `org.hibernate.cfg.Configuration`. This form is intended for use with the new metamodel code scheduled for completion in 5.0

See Section 7.7.1.1.2, "`org.hibernate.integrator.spi.IntegratorService`"

In addition to the discovery approach provided by the IntegratorService, applications can manually register Integrator implementations when building the BootstrapServiceRegistry. See Example 7.1, "Using BootstrapServiceRegistryBuilder"

7.9.1. Integrator use-cases

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 7.2. Registering event listeners

```
public class MyIntegrator implements org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            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( myDuplicationStrategy );

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH, myCompleteSetOfListeners );
        //     2) This form adds the specified listener(s) to the beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH, myListenersToBeCalledFirst );
        //     3) This form adds the specified listener(s) to the end of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH, myListenersToBeCalledLast );
    }
}
```

Chapter 8. Data categorizations

Table of Contents

Hibernate understands both the Java and JDBC representations of application data. The ability to read and write object data to a database is called *marshalling*, and is the function of a Hibernate `type`. A `type` is an implementation of the `org.hibernate.type.Type` interface. A Hibernate `type` describes various aspects of behavior of the Java type such as how to check for equality and how to clone values.

Usage of the word *type*

A Hibernate `type` is neither a Java type nor a SQL datatype. It provides information about both of these.

When you encounter the term *type* in regards to Hibernate, it may refer to the Java type, the JDBC type, or the Hibernate type, depending on context.

Hibernate categorizes types into two high-level groups: Section 8.1, “Value types” and Section 8.2, “Entity Types”.

8.1. Value types

A *value type* does not define its own lifecycle. It is, in effect, owned by an Section 8.2, “Entity Types”, which defines its lifecycle. Value types are further classified into three sub-categories.

- Section 8.1.1, “Basic types”
- Section 8.1.2, “Composite types”
- Section 8.1.3, “Collection types”

8.1.1. Basic types

Basic value types usually map a single database value, or column, to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which follow the natural mappings recommended in the JDBC specifications. You can override these mappings and provide and use alternative mappings. These topics are discussed further on.

Table 8.1. Basic Type Mappings

Hibernate type	Database type	JDBC type	Type registry
org.hibernate.type.StringType	string	VARCHAR	string, java.lang.String
org.hibernate.type.MaterializedClob	string	CLOB	materialized_clob
org.hibernate.type.TextType	string	LONGVARCHAR	text
org.hibernate.type.CharacterType	char, java.lang.Character	CHAR	char, java.lang.Character
org.hibernate.type.BooleanType	boolean	BIT	boolean, java.lang.Boolean

Hibernate type	Database type	JDBC type	Type registry
org.hibernate.type.NumericBooleanType	boolean	INTEGER, 0 is false, 1 is true	numeric_boolean
org.hibernate.type.YesNoType	boolean	CHAR, 'N'/'n' is false, 'Y'/'y' is true. The uppercase value is written to the database.	yes_no
org.hibernate.type.TrueFalseType	boolean	CHAR, 'F'/'f' is false, 'T'/'t' is true. The uppercase value is written to the database.	true_false
org.hibernate.type.ByteType	byte, java.lang.Byte	TINYINT	byte, java.lang.Byte
org.hibernate.type.ShortType	short, java.lang.Short	SMALLINT	short, java.lang.Short
org.hibernate.type.IntegerTypes	int, java.lang.Integer	INTEGER	int, java.lang.Integer
org.hibernate.type.LongType	long, java.lang.Long	BIGINT	long, java.lang.Long
org.hibernate.type.FloatType	float, java.lang.Float	FLOAT	float, java.lang.Float
org.hibernate.type.DoubleType	double, java.lang.Double	DOUBLE	double, java.lang.Double
org.hibernate.type.BigIntegerType	java.math.BigInteger	NUMERIC	big_integer
org.hibernate.type.BigDecimalType	java.math.BigDecimal	NUMERIC	big_decimal, java.math.bigDecimal
org.hibernate.type.TimestampType	java.sql.Timestamp	TIMESTAMP	timestamp, java.sql.Timestamp
org.hibernate.type.TimeType	java.sql.Time	TIME	time, java.sql.Time
org.hibernate.type.DateType	java.sql.Date	DATE	date, java.sql.Date
org.hibernate.type.CalendarType	java.util.Calendar	TIMESTAMP	calendar, java.util.Calendar
org.hibernate.type.CalendarDateType	java.util.Calendar	DATE	calendar_date
org.hibernate.type.CurrencyType	java.util.Currency	VARCHAR	currency, java.util.Currency
org.hibernate.type.LocaleType	java.util.Locale	VARCHAR	locale, java.utility.locale
org.hibernate.type.TimeZoneType	java.util.TimeZone	VARCHAR, using the TimeZone ID	timezone, java.util.TimeZone
org.hibernate.type.UrlType	java.net.URL	VARCHAR	url, java.net.URL
org.hibernate.type.ClassType	java.lang.Class	VARCHAR, using the class name	class, java.lang.Class
org.hibernate.type.BlobType	java.sql.Blob	BLOB	blog, java.sql.Blob

Hibernate type	Database type	JDBC type	Type registry
org.hibernate.type.ClobType	java.sql.Clob	CLOB	clob, java.sql.Clob
org.hibernate.type.BinaryType	primitive byte[]	VARBINARY	binary, byte[]
org.hibernate.type.MaterializedBlobType	primitive byte[]	BLOB	materized_blob
org.hibernate.type.ImageType	primitive byte[]	LONGVARBINARY	image
org.hibernate.type.BinaryType	java.lang.Byte[]	VARBINARY	wrapper-binary
org.hibernate.type.CharArrayType	char[]	VARCHAR	characters, char[]
org.hibernate.type.CharacterArrayType	java.lang.Character[]	VARCHAR	wrapper-characters, Character[], java.lang.Character[]
org.hibernate.type.UUIDBinaryType	java.util.UUID	BINARY	uuid-binary, java.util.UUID
org.hibernate.type.UUIDCharType	java.util.UUID	CHAR, can also read VARCHAR	uuid-char
org.hibernate.type.PostgresUUIDType	java.util.UUID	PostgreSQL UUID, through Types#OTHER, which complies to the PostgreSQL JDBC driver definition	pg-uuid
org.hibernate.type.SerializableType	implementors of java.lang.Serializable	VARBINARY	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.

8.1.2. Composite types

Composite types, or *embedded types*, as they are called by the Java Persistence API, have traditionally been called *components* in Hibernate. All of these terms mean the same thing.

Components represent aggregations of values into a single Java type. An example is an `Address` class, which aggregates street, city, state, and postal code. A composite type behaves in a similar way to an entity. They are each classes written specifically for an application. They may both include references to other application-specific classes, as well as to collections and simple JDK types. The only distinguishing factors are that a component does not have its own lifecycle or define an identifier.

8.1.3. Collection types

A *collection* type refers to the data type itself, not its contents.

A Collection denotes a one-to-one or one-to-many relationship between tables of a database.

Refer to the chapter on Collections for more information on collections.

8.2. Entity Types

Entities are application-specific classes which correlate to rows in a table, using a unique identifier. Because of the requirement for a unique identifier, ntities exist independently and define their own lifecycle. As an example, deleting a Membership should not delete the User or the Group. For more information, see the chapter on Persistent Classes.

8.3. Implications of different data categorizations

NEEDS TO BE WRITTEN

Chapter 9. Mapping entities

Table of Contents

9.1. Hierarchies

Chapter 10. Mapping associations

The most basic form of mapping in Hibernate is mapping a persistent entity class to a database table. You can expand on this concept by mapping associated classes together. ??? shows a `Person` class with a

Chapter 11. HQL and JPQL

Table of Contents

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 ??? for more information.

11.1. 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`.

Note

This documentation uses lowercase keywords as convention in examples.

11.2. 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-SELECT`.

Important

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*

11.2.1. Select statements

The BNF for `SELECT` statements in HQL is:

```
select_statement :: =
        [select_clause]
        from_clause
        [where_clause]
        [groupby_clause]
        [having_clause]
        [orderby_clause]
```

The simplest possible HQL `SELECT` statement is of the form:

```
from com.acme.Cat
```

The select statement in JPQL is exactly the same as for HQL except that JPQL requires a `select_clause`, whereas HQL does not. 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.

11.2.2. 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
```

`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. Note, however, that 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` 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 11.1. Example UPDATE query statements

```
String hqlUpdate =
                "update Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = session.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
String jpqlUpdate =
                "update Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = entityManager.createQuery( jpqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
String hqlVersionedUpdate =
                "update versioned Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
```

Important

Neither `UPDATE` nor `DELETE` statements are allowed to result in what is called an implicit join. Their form already disallows explicit joins.

11.2.3. 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]
```

A `DELETE` statement is also executed using the `executeUpdate` method of either `org.hibernate.Query` or `javax.persistence.Query`.

11.2.4. 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:

```
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 11.2. Example INSERT query statements

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert ).executeUpdate();
```

11.3. 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.

11.3.1. 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 thoughout 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.

11.3.2. 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 discussed later.

The BNF for a root entity reference is:

```
root_entity_reference ::= entity_name [AS] identification_variable
```

Example 11.3. Simple query example

```
select c from com.acme.Cat c
```

We see that the query is defining a root entity reference to the `com.acme.Cat` object model type. Additionally, it declares an alias of `c` to that `com.acme.Cat` reference; this is the identification variable.

Usually the root entity reference just names the `entity name` rather than the entity class FQN. By default the entity name is the unqualified entity class name, here `Cat`

Example 11.4. Simple query using entity name for root entity reference

```
select c from Cat c
```

Multiple root entity references can also be specified. Even naming the same entity!

Example 11.5. Simple query using multiple root entity references

```
// build a product between customers and active mailing campaigns so we can spam!
select distinct cust, camp
from Customer cust, Campaign camp
where camp.type = 'mail'
  and current_timestamp() between camp.activeRange.start and camp.activeRange.end
// retrieve all customers with headquarters in the same state as Acme's headquarters
select distinct c1
from Customer c1, Customer c2
where c1.address.state = c2.address.state
  and c2.name = 'Acme'
```

11.3.3. 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 11.6. Explicit inner join examples

```
select c
from Customer c
    join c.chiefExecutive ceo
where ceo.age < 25
```

```
// same query but specifying join type as 'inner' explicitly
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
```

Example 11.7. Explicit left (outer) join examples

```
// get customers who have orders worth more than $5000
// or who are in "preferred" status
select distinct c
from Customer c
    left join c.orders o
where o.value > 5000.00
  or c.status = 'preferred'

// functionally the same query but using the
// 'left outer' phrase
select distinct c
from Customer c
    left outer join c.orders o
where o.value > 5000.00
  or c.status = 'preferred'
```

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 `Customer` with a collection-valued association named `orders`

Example 11.8. Fetch join example

```
select c
from Customer c
    left join fetch c.orders o
```

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 want to return customers who have no orders also. Inner joins can also be fetched. But inner joins still filter. In the example, using an inner join instead would have resulted in customers without any orders being filtered out of the result.

Important

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 to not 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 (aka, `setFirstResult`/ `setMaxResults`). Nor should they be used with the HQL `scroll` or `iterate` features.

HQL also defines a `WITH` clause to qualify the join conditions. Again, this is specific to HQL; JPQL does not define this feature.

Example 11.9. with-clause join example

```
select distinct c
from Customer c
    left join c.orders o
        with o.value > 5000.00
```

The important distinction is that in the generated SQL the conditions of the `with 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 in more complicated queries.

Explicit joins may reference association or component/embedded attributes. For further information about collection-valued association references, see Section 11.3.5, “Collection member references”. In the case of component/embedded attributes, the join is simply logical and does not correlate to a physical (SQL) join.

11.3.4. 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 11.10. Simple implicit join example

```
select c
from Customer c
where c.chiefExecutive.age < 25

// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
```

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 `c` which refers to the `Customer` entity. The `c.chiefExecutive` reference then refers to the `chiefExecutive` attribute of the `Customer` entity. `chiefExecutive` is an association type so we further navigate to its `age` attribute.

Important

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 11.11. Reused implicit join

```
select c
from Customer c
where c.chiefExecutive.age < 25
   and c.chiefExecutive.address.state = 'TX'
```

```
// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
  and ceo.address.state = 'TX'

// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
    inner join ceo.address a
where ceo.age < 25
  and a.state = 'TX'
```

Just as with explicit joins, implicit joins may reference association or component/embedded attributes. For further information about collection-valued association references, see Section 11.3.5, "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.

11.3.5. Collection member references

References to collection-valued associations actually refer to the *values* of that collection.

Example 11.12. Collection references example

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
```

```
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

In the example, the identification variable `o` actually refers to the object model type `Order` which is the type of the elements of the `Customer#orders` 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.

11.3.5.1. 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 11.13. Qualified collection references example

```
// Product.images is a Map<String,String> : key = a name, value = file path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
```

```
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
        join c.orders o
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

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, can be further navigated.

ENTRY

Only valid only 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 only valid in the select clause.

See Section 11.4.9, "Collection-related expressions" for additional details on collection related expressions.

11.3.6. Polymorphism

HQL and JPQL queries are inherently polymorphic.

```
select p from Payment p
```

This query names the `Payment` entity explicitly. However, all subclasses of `Payment` are also available to the query. So if the `CreditCardPayment` entity and `WireTransferPayment` entity each extend from `Payment` all three types would be available to the query. And the query would return instances of all three.

The logical extreme

The HQL query `from java.lang.Object` is totally valid! It returns every object of every type defined in your application.

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.

11.4. Expressions

Essentially expressions are references that resolve to basic or tuple values.

11.4.1. Identification variable

See Section 11.3, "The `FROM` clause".

11.4.2. Path expressions

Again, see Section 11.3, "The `FROM` clause".

11.4.3. Literals

String literals are enclosed in single-quotes. To escape a single-quote within a string literal, use double single-quotes.

Example 11.14. String literal examples

```
select c
from Customer c
where c.name = 'Acme'

select c
from Customer c
where c.name = 'Acme''s Pretzel Logic'
```

Numeric literals are allowed in a few different forms.

Example 11.15. Numeric literal examples

```
// simple integer literal
select o
from Order o
where o.referenceNumber = 123

// simple integer literal, typed as a long
select o
from Order o
where o.referenceNumber = 123L

// decimal notation
select o
from Order o
where o.total > 5000.00

// decimal notation, typed as a float
```

```
select o
from Order o
where o.total > 5000.00F

// scientific notation
select o
from Order o
where o.total > 5e+3

// scientific notation, typed as a float
select o
from Order o
where o.total > 5e+3F
```

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 Section 11.4.10, "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 and `{ts 'yyyy-mm-dd hh:mm:ss[.millis]'}` (millis optional) for timestamps. These literals only work if you JDBC drivers supports them.

11.4.4. Parameters

HQL supports all 3 of the following forms. JPQL does not support the HQL-specific positional parameters notion. It is good practice to not mix forms in a given query.

11.4.4.1. Named parameters

Named parameters are declared using a colon followed by an identifier - `:aNamedParameter`. The same named parameter can appear multiple times in a query.

Example 11.16. Named parameter examples

```
String queryString =
        "select c " +
        "from Customer c " +
        "where c.name = :name " +
        "   or c.nickName = :name";

// HQL
List customers = session.createQuery( queryString )
        .setParameter( "name", theNameOfInterest )
        .list();

// JPQL
List<Customer> customers = entityManager.createQuery( queryString, Customer.class )
        .setParameter( "name", theNameOfInterest )
        .getResultList();
```

11.4.4.2. Positional (JPQL) parameters

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 11.17. Positional (JPQL) parameter examples

```
String queryString =
        "select c " +
        "from Customer c " +
        "where c.name = ?1 " +
        "   or c.nickName = ?1";
```

```
// HQL - as you can see, handled just like named parameters
//      in terms of API
List customers = session.createQuery( queryString )
        .setParameter( "1", theNameOfInterest )
        .list();

// JPQL
List<Customer> customers = entityManager.createQuery( queryString, Customer.class )
        .setParameter( 1, theNameOfInterest )
        .getResultList();
```

11.4.4.3. Positional (HQL) parameters

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.

This form should be considered deprecated and may be removed in the near future.

11.4.5. Arithmetic

Arithmetic operations also represent valid expressions.

Example 11.18. Numeric arithmetic examples

```
select year( current_date() ) - year( c.dateOfBirth )
from Customer c

select c
from Customer c
where year( current_date() ) - year( c.dateOfBirth ) < 30

select o.customer, o.total + ( o.total * :salesTax )
from Order o
```

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.

11.4.6. 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 - `||`.

Example 11.19. Concatenation operation example

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

See Section 11.4.8, “Scalar functions” for details on the `concat()` function

11.4.7. 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 `avg()` function depends on the type of the values being averaged. 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 11.20. Aggregate function examples

```
select count(*), sum( o.total ), avg( o.total ), min( o.total ), max( o.total )
from Order o

select count( distinct c.name )
from Customer c

select c.id, c.name, sum( o.total )
from Customer c
    left join c.orders o
group by c.id, c.name
```

Aggregations often appear with grouping. For information on grouping see Section 11.8, "Grouping"

11.4.8. 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.

11.4.8.1. Standardized functions - JPQL

Here are 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.

SUBSTRING

Extracts a portion of a string value.

```
substring( string_expression, numeric_expression [, numeric_expression] )
```

The second argument denotes the starting position. The third (optional) argument denotes the length.

UPPER

Upper cases the specified string

LOWER

Lower cases the specified string

TRIM

Follows the semantics of the SQL trim function.

LENGTH

Returns the length of a string.

LOCATE

Locates a string within another string.

```
locate( string_expression, string_expression[, numeric_expression] )
```

The third argument (optional) is used to denote a position from which to start looking.

ABS

Calculates the mathematical absolute value of a numeric value.

MOD

Calculates the remainder of dividing the first argument by the second.

SQRT

Calculates the mathematical square root of a numeric value.

CURRENT_DATE

Returns the database current date.

CURRENT_TIME

Returns the database current time.

CURRENT_TIMESTAMP

Returns the database current timestamp.

11.4.8.2. Standardized functions - HQL

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.

CAST

Performs a SQL cast. The cast target should name the Hibernate mapping type to use. See the chapter on data types for more information.

EXTRACT

Performs a SQL extraction on datetime values. An extraction extracts parts of the datetime (the year, for example). See the abbreviated forms below.

SECOND

Abbreviated extract form for extracting the second.

MINUTE

Abbreviated extract form for extracting the minute.

HOUR

Abbreviated extract form for extracting the hour.

DAY

Abbreviated extract form for extracting the day.

MONTH

Abbreviated extract form for extracting the month.

YEAR

Abbreviated extract form for extracting the year.

STR

Abbreviated form for casting a value as character data.

11.4.8.3. 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`

11.4.9. 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 11.21. Collection-related expressions examples

```
select cal
from Calendar cal
where maxelement(cal.holidays) > current_date()

select o
from Order o
where maxindex(o.items) > 100

select o
from Order o
where minelement(o.items) > 10000

select m
from Cat as m, Cat as kit
where kit in elements(m.kittens)

// the above query can be re-written in jpql standard way:
select m
from Cat as m, Cat as kit
where kit member of m.kittens

select p
from NameList l, Person p
```

```
where p.name = some elements(l.names)

select cat
from Cat cat
where exists elements(cat.kittens)

select p
from Player p
where 3 > all elements(p.scores)

select show
from Show show
where 'fizard' in indices(show.acts)
```

Elements of indexed collections (arrays, lists, and maps) can be referred to by index operator.

Example 11.22. Index operator examples

```
select o
from Order o
where o.items[0].id = 1234

select p
from Person p, Calendar c
where c.holidays['national day'] = p.birthDay
  and p.nationality.calendar = c

select i
from Item i, Order o
where o.items[ o.deliveredItemIndices[0] ] = i
  and o.id = 11

select i
from Item i, Order o
where o.items[ maxindex(o.items) ] = i
  and o.id = 11

select i
```

```
from Item i, Order o
where o.items[ size(o.items) - 1 ] = i
```

See also Section 11.3.5.1, “Special case - qualified path expressions” as there is a good deal of overlap.

11.4.10. 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 parametrized, in which case the entity's Java Class reference would be bound as the parameter value.

Example 11.23. Entity type expression examples

```
select p
from Payment p
where type(p) = CreditCardPayment

select p
from Payment p
where type(p) = :aType
```

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.

11.4.11. CASE expressions

Both the simple and searched forms are supported, as well as the 2 SQL defined abbreviated forms (`NULLIF` and `COALESCE`)

11.4.11.1. Simple CASE expressions

The simple form has the following syntax:

```
CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} END
```

Example 11.24. Simple case expression example

```
select case c.nickName when null then '<no nick name>' else c.nickName end
from Customer c

// This NULL checking is such a common case that most dbs
// define an abbreviated CASE form.  For example:
select nvl( c.nickName, '<no nick name>' )
from Customer c

// or:
select isnull( c.nickName, '<no nick name>' )
from Customer c

// the standard coalesce abbreviated form can be used
// to achieve the same result:
select coalesce( c.nickName, '<no nick name>' )
from Customer c
```

11.4.11.2. Searched CASE expressions

The searched form has the following syntax:

```
CASE [ WHEN {test_conditional} THEN {match_result} ]* ELSE {miss_result} END
```

Example 11.25. Searched case expression example

```
select case when c.name.first is not null then c.name.first
            when c.nickName is not null then c.nickName
            else '<no first name>' end
```

```
from Customer c

// Again, the abbreviated form coalesce can handle this a
// little more succinctly
select coalesce( c.name.first, c.nickName, '<no first name>' )
from Customer c
```

11.4.11.3. NULLIF expressions

NULLIF is an abbreviated CASE expression that returns NULL if its operands are considered equal.

Example 11.26. NULLIF example

```
// return customers who have changed their last name
select nullif( c.previousName.last, c.name.last )
from Customer c

// equivalent CASE expression
select case when c.previousName.last = c.name.last then null
            else c.previousName.last end
from Customer c
```

11.4.11.4. COALESCE expressions

COALESCE is an abbreviated CASE expression that returns the first non-null operand. We have seen a number of COALESCE examples above.

11.5. The `SELECT` clause

The `SELECT` clause identifies which objects and values to return as the query results. The expressions discussed in Section 11.4, “Expressions” are all valid select expressions, except where otherwise noted. See the section Section 11.10, “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”

Example 11.27. Dynamic instantiation example - constructor

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

So rather than dealing with the Object[] (again, see Section 11.10, “Query API”) here we are wrapping the values in a type-safe java object that will be returned as the results of the query. 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!).

That is the part JPQL supports as well. HQL supports additional “dynamic instantiation” features. First, the query can specify to return a List rather than an Object[] for scalar results:

Example 11.28. Dynamic instantiation example - list

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

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 11.29. Dynamic instantiation example - map

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min, count(*) as n )
from Cat c
```

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.

11.6. 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 NULLs generally resolve to `UNKNOWN`.

11.6.1. Relational comparisons

Comparisons involve one of the comparison operators - =, >, >=, <, <=, <>]>. HQL also defines <![CDATA[!= as a comparison operator synonymous with <>. The operands should be of the same type.

Example 11.30. Relational comparison examples

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
```

```
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

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 11.31. ALL subquery comparison qualifier example

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
    select spg.points
    from StatsPerGame spg
    where spg.player = p
)
```

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.

11.6.2. 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 11.32. Nullness checking examples

```
// select everyone with an associated address
select p
from Person p
where p.address is not null

// select everyone without an associated address
select p
from Person p
where p.address is null
```

11.6.3. Like predicate

Performs a like comparison on string values. The syntax is:

```
like_expression ::=
        string_expression
        [NOT] LIKE pattern_value
        [ESCAPE escape_character]
```

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. "_" matches any single character. "%" matches any number of characters.

The optional `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 "%"

Example 11.33. Like predicate examples

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimmer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

11.6.4. Between predicate

Analogous to the SQL between expression. Perform a evaluation that a value is within the range of 2 other values. All the operands should have comparable types.

Example 11.34. Between predicate examples

```
select p
from Customer c
    join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9
```

```
select c
from Customer c
where c.president.dateOfBirth
        between {d '1945-01-01'}
            and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'
```

11.6.5. In predicate

`IN` predicates performs a check that a particular value is in a list of values. Its syntax is:

```
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. 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 Section 11.4.10, “Entity type”

In HQL, `single_valued_expression` can refer to a far more broad set of expression types. Single-valued association are allowed. 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 11.35. In predicate examples

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not JPQL compliant!
select c
```

```
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

11.6.6. 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.

11.6.7. 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 11.36. Empty collection expression examples

```
select o
from Order o
where o.lineItems is empty

select c
from Customer c
where c.pastDueBills is not empty
```

11.6.8. 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 11.37. Member-of collection expression examples

```
select p
from Person p
where 'John' member of p.nickNames

select p
from Person p
where p.name.first = 'Joseph'
  and 'Joey' not member of p.nickNames
```

11.6.9. 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, the NOT resolves to unknown as well.

11.6.10. 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 predicate resolves to unknown, the AND expression resolves to unknown as well. Otherwise, the result is false.

11.6.11. OR predicate operator

The `OR` operator is used to combine 2 predicate expressions. The result of the OR expression is true if either predicate resolves to true. If both predicates resolve to unknown, the OR expression resolves to unknown. Otherwise, the result is false.

11.7. The `WHERE` clause

The `WHERE` clause of a query is made up of predicates which assert whether values in each potential row match the predicated checks. Thus, the where clause restricts the results returned from a select query and limits the scope of update and delete queries.

11.8. Grouping

The `GROUP BY` clause allows building aggregated results for various value groups. As an example, consider the following queries:

Example 11.38. Group-by illustration

```
// retrieve the total for all orders
select sum( o.total )
from Order o

// retrieve the total of all orders
// *grouped by* customer
select c.id, sum( o.total )
from Order o
    inner join o.customer c
group by c.id
```

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 Example 11.38, “Group-by illustration” example, we retrieved order totals for all customers. 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 order total of more than $10,000.00:

Example 11.39. Having illustration

```
select c.id, sum( o.total )
from Order o
    inner join o.customer c
group by c.id
having sum( o.total ) > 10000.00
```

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; WHERE is applied before.

11.9. Ordering

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.

Example 11.40. Order-by examples

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
    inner join o.customer c
group by c.id
order by t
```

11.10. Query API

Chapter 12. Criteria

Table of Contents

Criteria queries offer a type-safe alternative to HQL, JPQL and native-sql queries.

Important

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 ???.

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, or 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 3 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.

Note

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, lets instead look at some of the more widely anticipated usages of the API.

12.1. 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.

12.1.1. Selecting an entity

This is probably the most common form of query. The application wants to select entity instances.

Example 12.1. Selecting the root entity

```
CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Person> people = em.createQuery( criteria ).getResultList();
for ( Person person : people ) {
    ...
}
```

The example uses `createQuery` passing in the `Person` class reference as the results of the query will be Person objects.

Note

The call to the `CriteriaQuery.select` method in this example is unnecessary because *personRoot* 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_.eyeColor* 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 for additional details on the JPA static metamodel.

12.1.2. 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 12.2. Selecting an attribute

```
CriteriaQuery<Integer> criteria = builder.createQuery( Integer.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot.get( Person_.age ) );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Integer> ages = em.createQuery( criteria ).getResultList();
```

```
for ( Integer age : ages ) {
        ...
}
```

In this example, the query is typed as `java.lang.Integer` because that is the anticipated type of the results (the type of the `Person#age` attribute is `java.lang.Integer`). 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.

12.1.3. Selecting multiple values

There are actually a few different ways to select multiple values using criteria queries. We will explore 2 options here, but an alternative recommended approach is to use tuples as described in Section 12.2, "Tuple criteria queries". Or consider a wrapper query; see Section 12.1.4, "Selecting a wrapper" for details.

Example 12.3. Selecting an array

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.select( builder.array( idPath, agePath ) );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();
for ( Object[] values : valueArray ) {
        final Long id = (Long) values[0];
        final Integer age = (Integer) values[1];
        ...
}
```

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 12.4. Selecting an array (2)

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.multiselect( idPath, agePath );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();
for ( Object[] values : valueArray ) {
    final Long id = (Long) values[0];
    final Integer age = (Integer) values[1];
    ...
}
```

Just as we saw in Example 12.3, “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[]*.

12.1.4. Selecting a wrapper

Another alternative to Section 12.1.3, “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#age]* instead declare a class that holds these values and instead return that.

Example 12.5. Selecting an wrapper

```
public class PersonWrapper {
        private final Long id;
        private final Integer age;
        public PersonWrapper(Long id, Integer age) {
                this.id = id;
                this.age = age;
        }
        ...
}
```

```
...

CriteriaQuery<PersonWrapper> criteria = builder.createQuery( PersonWrapper.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select(
               builder.construct(
                       PersonWrapper.class,
                       personRoot.get( Person_.id ),
                       personRoot.get( Person_.age )
               )
);
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<PersonWrapper> people = em.createQuery( criteria ).getResultList();
for ( PersonWrapper person : people ) {
    ...
}
```

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.

12.2. Tuple criteria queries

A better approach to Section 12.1.3, “Selecting multiple values” is to use either a wrapper (which we just saw in Section 12.1.4, “Selecting a wrapper”) or using the `javax.persistence.Tuple` contract.

Example 12.6. Selecting a tuple

```
CriteriaQuery<Tuple> criteria = builder.createTupleQuery();
Root<Person> personRoot = criteria.from( Person.class );
```

```
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.multiselect( idPath, agePath );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Tuple> tuples = em.createQuery( criteria ).getResultList();
for ( Tuple tuple : valueArray ) {
    assert tuple.get( 0 ) == tuple.get( idPath );
        assert tuple.get( 1 ) == tuple.get( agePath );
        ...
}
```

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` passing `Tuple.class`.

Again we see the use of the `multiselect` method, just like in Example 12.4, “Selecting an array (2)”. 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 3 forms of access to the underlying elements:

typed

The Example 12.6, “Selecting a tuple” example illustrates this form of access in the `tuple.get( idPath )` and `tuple.get( agePath )` 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 Example 12.3, “Selecting an array” and Example 12.4, “Selecting an array (2)”. 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.

12.3. 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

Note

All the individual parts of the FROM clause (roots, joins, paths) implement the `javax.persistence.criteria.From` interface.

12.3.1. 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`:

```
<X> Root<X> from(Class<X>);
```

```
<X> Root<X> from(EntityType<X>)
```

Example 12.7. Adding a root

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
// create and add the root
person.from( Person.class );
```

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 matching all single men and all single women:

Example 12.8. Adding multiple roots

```
CriteriaQuery query = builder.createQuery();
Root<Person> men = query.from( Person.class );
Root<Person> women = query.from( Person.class );
Predicate menRestriction = builder.and(
               builder.equal( men.get( Person_.gender ), Gender.MALE ),
               builder.equal( men.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
);
Predicate womenRestriction = builder.and(
               builder.equal( women.get( Person_.gender ), Gender.FEMALE ),
               builder.equal( women.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
);
query.where( builder.and( menRestriction, womenRestriction ) );
```

12.3.2. 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 12.9. Example with Embedded and ManyToOne

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Join<Person,Address> personAddress = personRoot.join( Person_.address );
// Address.country is a ManyToOne
Join<Address,Country> addressCountry = personAddress.join( Address_.country );
```

Example 12.10. Example with Collections

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Join<Person,Order> orders = personRoot.join( Person_.orders );
Join<Order,LineItem> orderLines = orders.join( Order_.lineItems );
```

12.3.3. 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 12.11. Example with Embedded and ManyToOne

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Fetch<Person,Address> personAddress = personRoot.fetch( Person_.address );
// Address.country is a ManyToOne
Fetch<Address,Country> addressCountry = personAddress.fetch( Address_.country );
```

Note

Technically speaking, embedded attributes are always fetched with their owner. However in order to define the fetching of *Address#country* we needed a `javax.persistence.criteria.Fetch` for its parent path.

Example 12.12. Example with Collections

```
CriteriaQuery&lt;Person&gt; personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Fetch<Person,Order> orders = personRoot.fetch( Person_.orders );
```

```
Fetch<Order,LineItem> orderLines = orders.fetch( Order_.lineItems );
```

12.4. Path expressions

Note

Roots, joins and fetches are themselves paths as well.

12.5. Using parameters

Example 12.13. Using parameters

```
CriteriaQuery<Person> criteria = build.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot );
ParameterExpression<String> eyeColorParam = builder.parameter( String.class );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), eyeColorParam ) );

TypedQuery<Person> query = em.createQuery( criteria );
query.setParameter( eyeColorParam, "brown" );
List<Person> people = 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`

Chapter 13. Native SQL Queries

Table of Contents

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 query hints 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 load operations.

13.1. Using a `SQLQuery`

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.

13.1.1. Scalar queries

The most basic SQL query is to get a list of scalars (values).

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

These will return a List of Object arrays (Object[]) with scalar values for each column in the CATS table. Hibernate will use 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()`:

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

This query specified:

- the SQL query string
- the columns and types to return

This will return Object arrays, but now it will not use `ResultSetMetadata` but will instead explicitly get the ID, NAME and BIRTHDATE column as respectively a Long, String and a Short from the underlying resultset. This also means that only these three columns will be returned, even though the query is using `*` and could return more than the three listed columns.

It is possible to leave out the type information for all or some of the scalars.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

This is essentially the same query as before, but now `ResultSetMetaData` is used to determine the type of NAME and BIRTHDATE, 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.

13.1.2. Entity queries

The above queries were all about returning scalar values, basically returning the "raw" values from the resultset. The following shows how to get entity objects from a native sql query via `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

This query specified:

- the SQL query string
- the entity returned by the query

Assuming that Cat is mapped as a class with the columns ID, NAME and BIRTHDATE the above queries will both return a List where each element is a Cat entity.

If the entity is mapped with a `many-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. The additional columns will automatically be returned when using the * notation, but we prefer to be explicit as in the following example for a `many-to-one` to a `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

This will allow cat.getDog() to function properly.

13.1.3. Handling associations and collections

It is possible to eagerly join in the `Dog` to avoid the possible extra roundtrip for initializing the proxy. This is done via the `addJoin()` method, which allows you to join in an association or collection.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

In this example, the returned `Cat`'s will have their `dog` property fully initialized without any extra roundtrip to the database. Notice that you added an alias name ("cat") 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. if the `Cat` had a one-to-many to `Dog` instead.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

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.

13.1.4. 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 most likely will fail):

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

The query was intended to return two Cat instances per row: a cat and its mother. The query will, however, fail because there is a conflict of names; the instances are mapped to the same column names. Also, on some databases the returned column aliases will most likely be on the form "c.ID", "c.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:

```
sess.createSQLQuery("SELECT {cat.*}, {m.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = m.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

This query specified:

- the SQL query string, with placeholders for Hibernate to inject column aliases
- the entities returned by the query

The {cat.*} and {mother.*} notation used above is a 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. In the following example, you retrieve Cats and their mothers from a different table (cat_log) to the one declared in the mapping metadata. You can even use the property aliases in the where clause.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

13.1.4.1. 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 13.1. 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}`
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.*}`

13.1.5. Returning non-managed entities

It is possible to apply a ResultTransformer to native SQL queries, allowing it to return non-managed entities.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

This query specified:

- the SQL query string
- a result transformer

The above query will return a list of `CatDTO` which has been instantiated and injected the values of NAME and BIRTHNAME into its corresponding properties or fields.

13.1.6. Handling inheritance

Native SQL queries which query for entities that are mapped as part of an inheritance must include all properties for the baseclass and all its subclasses.

13.1.7. Parameters

Native SQL queries support positional as well as named parameters:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

13.2. Named SQL queries

Named SQL queries can also be defined in the mapping document and called in exactly the same way as a named HQL query (see ???). In this case, you do *not* need to call `addEntity()`.

Example 13.1. Named sql query using the <sql-query> maping element

```
<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 13.2. Execution of a named query

```
List people = sess.getNamedQuery("persons")
    .setString("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 13.3. Named sql query with association

```
<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 13.4. Named query returning a scalar

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

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 13.5. <resultset> mapping used to externalize mapping information

```
<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>
```

You can, alternatively, use the resultset mapping information in your hbm files directly in java code.

Example 13.6. Programmatically specifying the result mapping information

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

So far we have only looked at externalizing SQL queries using Hibernate mapping files. The same concept is also available with anntations and is called named native queries. You can use `@NamedNativeQuery` (`@NamedNativeQueries`) in conjunction with `@SqlResultSetMapping` (`@SqlResultSetMappings`). Like `@NamedQuery`, `@NamedNativeQuery` and `@SqlResultSetMapping` can be defined at class level, but their scope is global to the application. Lets look at a view examples.

Example 13.7, "Named SQL query using `@NamedNativeQuery` together with `@SqlResultSetMapping`" shows how a `resultSetMapping` parameter is defined in `@NamedNativeQuery`. It represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property. In the example 2 entities, `Night` and `Area`, are returned and each property is declared and associated to a column name, actually the column name retrieved by the query.

In Example 13.8, "Implicit result set mapping" the result set mapping is implicit. We only describe the entity class of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column.

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. This can be seen in Example 13.9, “Using dot notation in @FieldResult for specifying associations ”.

Example 13.7. Named SQL query using `@NamedNativeQuery` together with `@SqlResultSetMapping`

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

Example 13.8. Implicit result set mapping

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

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

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

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

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

    public double getSpeed() {
        return speed;
    }

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

Example 13.9. Using dot notation in @FieldResult for specifying associations

```
@Entity
```

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

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

private Captain captain;

private Dimensions dimensions;


@Id

public String getName() {

    return name;

}


public void setName(String name) {

    this.name = name;

}


@ManyToOne(fetch= FetchType.LAZY)

@JoinColumns( {

        @JoinColumn(name="fname", referencedColumnName = "firstname"),

        @JoinColumn(name="lname", referencedColumnName = "lastname")

        } )

public Captain getCaptain() {

    return captain;

}
```

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

public String getModel() {
    return model;
}

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

public double getSpeed() {
    return speed;
}

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

public Dimensions getDimensions() {
    return dimensions;
}
```

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

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

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

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

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

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

Tip

If you retrieve a single entity using the default mapping, you can specify the `resultClass` attribute instead of `resultSetMapping`:

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

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

Example 13.10. Scalar values via `@ColumnResult`

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

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

13.2.1. Using 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:

```
<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.

```
<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.

13.2.2. Using stored procedures for querying

Hibernate3 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:

```
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.

```
<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.

13.2.2.1. 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.connection()`. 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.

13.3. Custom SQL for create, update and delete

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or inidividual column level. This section describes statement overrides. For columns, see ???. Example 13.11, “Custom CRUD via annotations” shows how to define custom SQL operatons using annotations.

Example 13.11. Custom CRUD via annotations

```
@Entity

@Table(name="CHAOS")
```

```
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")

@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")

@SQLDelete( sql="DELETE CHAOS WHERE id = ?")

@SQLDeleteAll( sql="DELETE CHAOS")

@Loader(namedQuery = "chaos")

@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from CHAOS where id= ?", resultClass = Chaos.class)

public class Chaos {

    @Id

    private Long id;

    private Long size;

    private String name;

    private String nickname;
```

`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`, `@SQLDeleteAll` respectively override the INSERT, UPDATE, DELETE, and DELETE all statement. The same can be achieved using Hibernate mapping files and the `<sql-insert>`, `<sql-update>` and `<sql-delete>` nodes. This can be seen in Example 13.12, “Custom CRUD XML”.

Example 13.12. Custom CRUD 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 annoations as well as in xml.

You can use the exact same set of annotations respectively xml nodes to override the collection related statements -see Example 13.13, “Overriding SQL statements for collections using annotations”.

Example 13.13. Overriding SQL statements for collections using annotations

```
@OneToMany

@JoinColumn(name="chaos_fk")

@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")

@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")

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

Tip

The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations 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 either (or all) attributes `sqlInsert`, `sqlUpdate`, `sqlDelete`:

Example 13.14. Overriding SQL statements for secondary tables

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

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

Tip

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.

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 13.15. 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;
```

13.4. 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 ??? or at the statement level. Here is an example of a statement level override:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

This is just a named query declaration, as discussed earlier. You can reference this named query in a class mapping:

```
<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:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

You can also define an entity loader that loads a collection by join fetching:

```
<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>
```

The annotation equivalent `<loader>` is the @Loader annotation as seen in Example 13.11, “Custom CRUD via annotations”.

Chapter 14. JMX

Chapter 15. Envers

Abstract

The aim of Hibernate Envers is to provide historical versioning of your application's entity data. Much like source control management tools such as Subversion or Git, Hibernate Envers manages a notion of revisions if your application data through the use of audit tables. Each transaction relates to one global revision number which can be used to identify groups of changes (much like a change set in source control). As the revisions are global, having a revision number, you can query for various entities at that revision, retrieving a (partial) view of the database at that revision. You can find a revision number having a date, and the other way round, you can get the date at which a revision was committed.

Table of Contents

15.1. Basics

To audit changes that are performed on an entity, you only need two things: the `hibernate-envers` jar on the classpath and an `@Audited` annotation on the entity.

Important

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 - 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`. Otherwise, to export complete database schema programatically, use `org.hibernate.tool.EnversSchemaGenerator`. Appropriate DDL statements can be also generated with Ant task described later in this manual.

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 for these classes for details on the functionality offered.

15.2. Configuration

It is possible to configure various aspects of Hibernate Envers behavior, such as table names, etc.

Table 15.1. Envers Configuration Properties

Property name	Default value	Description
org.hibernate.envers.audit_table_prefix		String that will be prepended to the name of an audited entity to c name of the entity, that will hold audit information.
org.hibernate.envers.audit_table_suffix	_AUD	String that will be appended to the name of an audited entity to cr name of the entity, that will hold audit information. If you audit a a table name Person, in the default setting Envers will generate a table to store historical data.
org.hibernate.envers.revision_field_name	REV	Name of a field in the audit entity that will hold the revision num
org.hibernate.envers.revision_type_field_name	REVTYPE	Name of a field in the audit entity that will hold the type of the re (currently, this can be: add, mod, del).
org.hibernate.envers.revision_on_collection_change	true	Should a revision be generated when a not-owned relation field c can be either a collection in a one-to-many relation, or the field u "mappedBy" attribute in a one-to-one relation).
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	When true, properties to be used for optimistic locking, annotated `@Version`, will be automatically not audited (their history won't b normally doesn't make sense to store it).
org.hibernate.envers.store_data_at_delete	false	Should the entity data be stored in the revision when the entity is (instead of only storing the id and all other properties as null). Th normally needed, as the data is present in the last-but-one revisio Sometimes, however, it is easier and more efficient to access it in revision (then the data that the entity contained before deletion is twice).
org.hibernate.envers.default_schema	null (same schema as table being audited)	The default schema name that should be used for audit tables. Ca overridden using the `@AuditTable(schema="...")` annotation. I present, the schema will be the same as the schema of the table be
org.hibernate.envers.default_catalog	null (same catalog as table being audited)	The default catalog name that should be used for audit tables. Ca overridden using the `@AuditTable(catalog="...")` annotation. present, the catalog will be the same as the catalog of the normal

Property name	Default value	Description
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	The audit strategy that should be used when persisting audit data. stores only the revision, at which an entity was modified. An alter `org.hibernate.envers.strategy.ValidityAuditStrategy` st the start revision and the end revision. Together these define whe row was valid, hence the name ValidityAuditStrategy.
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	The column name that will hold the end revision number in audit property is only valid if the validity audit strategy is used.
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	Should the timestamp of the end revision be stored, until which th valid, in addition to the end revision itself. This is useful to be abl old Audit records out of a relational database by using table partit Partitioning requires a column that exists within the table. This pr only evaluated if the ValidityAuditStrategy is used.
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	Column name of the timestamp of the end revision until which the valid. Only used if the ValidityAuditStrategy is used, and org.hibernate.envers.audit_strategy_validity_store_revend_timest evaluates to true
org.hibernate.envers.use_revision_entity_with_native_id	true	Boolean flag that determines the strategy of revision number gene Default implementation of revision entity uses native identifier ge current database engine does not support identity columns, users a to set this property to false. In this case revision numbers are creat preconfigured `org.hibernate.id.enhanced.SequenceStyleGe` See: 1. `org.hibernate.envers.DefaultRevisionEntity` 2. `org.hibernate.envers.enhanced.SequenceIdRevisio`
org.hibernate.envers.track_entities_changed_in_revision	false	Should entity types, that have been modified during each revision The default implementation creates `REVCHANGES` table that stores of modified persistent objects. Single record encapsulates the revi identifier (foreign key to `REVINFO` table) and a string value. For m

Property name	Default value	Description
		information refer to Section 15.5.1, "Tracking entity names modif revisions" and Section 15.7.4, "Querying for entities modified in a revision".
org.hibernate.envers.global_with_modified_flag	false, can be individually overriden with `@Audited(withModifiedFlag=true)`	Should property modification flags be stored for all audited entitie properties. When set to true, for all properties an additional boolea the audit tables will be created, filled with information if the giver changed in the given revision. When set to false, such column can selected entities or properties using the `@Audited` annotation. For information refer to Section 15.6, "Tracking entity changes at pro and Section 15.7.3, "Querying for revisions of entity that modifiec property".
org.hibernate.envers.modified_flag_suffix	_MOD	The suffix for columns storing "Modified Flags". For example: a p called "age", will by default get modified flag with column name "age_MOD".

Important

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

15.3. 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 behaviour, you can use the `@SecondaryAuditTable` and `@SecondaryAuditTables` annotations.

If you'd like to override auditing behaviour 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+@JoinColumn`, please see Section 15.11, “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, when reading historic versions of your entity, the relation will always point to the "current" related entity.

If you'd like to audit properties of a superclass of an entity, which are not explicitly audited (which don't have the `@Audited` annotation on any properties or on the class), you can list the superclasses in the `auditParents` attribute of the `@Audited` annotation. Please note that `auditParents` feature has been deprecated. Use `@AuditOverride(forClass = SomeEntity.class, isAudited = true/false)` instead.

15.4. 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 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 by adding extra indexes.

15.5. Revision Log

Logging data for revisions

When Envers starts a new revision, it creates a new *revision entity* which stores information about the revision. By default, that includes just

1. *revision number* - An integral value (`int/Integer` or `long/Long`). Essentially the primary key of the revision
2. *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 2 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 2 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. Envers will "find it". Note that 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` method of the `org.jboss.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. exists in a different module), set org.hibernate.envers.revision_listener property to it's fully qualified name. Class name defined by the configuration parameter overrides revision entity's `value` attribute.

```
@Entity
@RevisionEntity( MyCustomRevisionListener.class )
public class MyCustomRevisionEntity {
    ...
}
```

```
public class MyCustomRevisionListener implements RevisionListener {
    public void newRevision(Object revisionEntity) {
        ( (MyCustomRevisionEntity) revisionEntity )...;
    }
}
```

An alternative method to using the `org.hibernate.envers.RevisionListener` is to instead call the `getCurrentRevision` 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.

Example 15.1. Example of storing username with revision

```
                ExampleRevEntity.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; }
}
                ExampleListener.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());
    }
}
```

15.5.1. 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.

   ```
   @Entity
   @RevisionEntity
   public class ExtendedRevisionEntity
                extends DefaultTrackingModifiedEntitiesRevisionEntity {
       ...
   }
   ```

8. 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.

   ```
   @Entity
    @RevisionEntity
    public class AnnotatedTrackingRevisionEntity {
        ...
   ```

```

    @ElementCollection
    @JoinTable(name = "REVCHANGES", joinColumns = @JoinColumn(name = "REV"))
    @Column(name = "ENTITYNAME")
    @ModifiedEntityNames
    private Set<String> modifiedEntityNames;

    ...
}
```

Users, that have chosen one of the approaches listed above, can retrieve all entities modified in a specified revision by utilizing API described in Section 15.7.4, “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 15.2. Custom implementation of tracking entity classes modified during revisions

```
                    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) {
    }
}
                    CustomTrackingRevisionEntity.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));
    }

    ...
}
```

ModifiedEntityTypeEntity.java

```
@Entity
public class ModifiedEntityTypeEntity {
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne
    private CustomTrackingRevisionEntity revision;

    private String entityClassName;

    ...
}
```

```
CustomTrackingRevisionEntity revEntity =
    getAuditReader().findRevision(CustomTrackingRevisionEntity.class, revisionNumber);
Set<ModifiedEntityTypeEntity> modifiedEntityTypes = revEntity.getModifiedEntityTypes()
```

15.6. 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's 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 Section 15.5.1, "Tracking entity names modified during revisions" makes it possible to tell which entities were modified in given revision. 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 for all audited properties in 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's 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: Section 15.7.3, "Querying for revisions of entity that modified given property".

15.7. Queries

You can think of historic data as having two dimension. The first - horizontal - is the state of the database at a given revision. Thus, you can query for entities as they were at revision N. The second - 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.

In the future, queries will be 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 ???.

15.7.1. Querying for entities of a class at a given revision

The entry point for this type of queries is:

```
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":

```
query.add(AuditEntity.property("name").eq("John"));
```

And to select only entites that are related to a given entity:

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

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:

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

15.7.2. Querying for revisions, at which entities of a given class changed

The entry point for this type of queries is:

```
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:

```
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 maximalize/minimize a property. For example, if you want to select the revision, at which the value of the `actualDate` for a given entity was larger then a given value, but as small as possible:

```
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.

You probably also noticed that there are two boolean parameters, passed when creating the query. The first one, `selectEntitiesOnly`, 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).

The second parameter, `selectDeletedEntities`, 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`.

15.7.3. 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 Section 15.6, “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:

```
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.

```
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 we 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:

```
AuditQuery query = getAuditReader().createQuery()
.forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
.add(AuditEntity.property("prop1").hasChanged())
.add(AuditEntity.property("prop2").hasNotChanged());
```

15.7.4. Querying for entities modified in a given revision

The basic query allows retrieving entity names and corresponding Java classes changed in a specified revision:

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

Other queries (also accessible from `org.hibernate.envers.CrossTypeRevisionChangesReader`):

1. *`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.
2. *`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.
3. *`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 default mechanism of tracking changed entity names is enabled (see Section 15.5.1, “Tracking entity names modified during revisions”).

15.8. 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 even listeners from the `org.hibernate.envers.event` 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.eventEnversPostInsertEventListenerImpl` 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.event.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.

15.9. 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 a "_AUD" suffix to the original table name, but this can be overridden by specifying a different suffix/prefix in the configuration or per-entity using the `@org.hibernate.envers.AuditTable` annotation.

Audit table columns

1. id of the original entity (this can be more then one column in the case of composite primary keys)
2. revision number - an integer. Matches to the revision number in the revision entity table.
3. revision type - a small integer
4. audited fields from the original entity

The primary key of the audit table is the combination of the original id of the entity and the revision number - 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, 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 2 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 Section 15.5, “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 wouldn't enable correct versioning of relations, but wouldn't also require the REVINFO table. Another possibility is to introduce a notion of "revisioning groups": groups of entities which share revision numbering. Each such group would have to consist of one or more strongly connected component of the graph induced by relations between entities. Your opinions on the subject are very welcome on the forum! :)

15.10. Generating schema with Ant

If you'd like to generate the database schema file with the Hibernate Tools Ant task, you'll probably notice that the generated file doesn't contain definitions of audit tables. To generate also the audit tables, you simply need to use `org.hibernate.tool.ant.EnversHibernateToolTask` instead of the usual `org.hibernate.tool.ant.HibernateToolTask`. The former class extends the latter, and only adds generation of the version entities. So you can use the task just as you used to.

For example:

```
<target name="schemaexport" depends="build-demo"
  description="Exports a generated schema to DB and file">
  <taskdef name="hibernatetool"
    classname="org.hibernate.tool.ant.EnversHibernateToolTask"
    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="versioning-ddl.sql"
      delimiter=";"
      format="true"/>
  </hibernatetool>
</target>
```

Will generate the following schema:

```
    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;
```

15.11. Mapping exceptions

15.11.1. What isn't and will not be supported

Bags (the corresponding Java type is List), as they can contain non-unique elements. The reason is that persisting, for example a bag of String-s, violates a principle of relational databases: 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 `@IndexColumn` annotation, or
2. provide a unique id for your elements with the `@CollectionId` annotation.

15.11.2. What isn't and *will* be supported

1. collections of components

15.11.3. `@OneToMany`+`@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's `@JoinTable`.

One special case are relations mapped with `@OneToMany`+`@JoinColumn` on the one side, and `@ManyToOne`+`@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.

15.12. Advanced: Audit table partitioning

15.12.1. 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.

15.12.2. 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 partioning of audit tables.

> **Note**
>
> 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 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 ???.

The reason why the end revision information should be used for audit table partioning is based on the assumption that audit tables should be partionioned on an 'increasing level of interestingness', like so:

1. A couple of partitions with audit data that is not very (or no longer) interesting. This can be stored on slow media, and perhaps even be purged eventually.
2. Some partitions for audit data that is potentially interesting.
3. One partition for audit data that is most likely to be interesting. This should be stored on the fastest media, both for reading and writing.

15.12.3. Audit table partitioning example

In order to determine a suitable column for the 'increasing level of interestingness', 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 15.2. Salaries table

Year	Salary (USD)
2006	3300
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 15.3. 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

Year	Revision type	Revision timestamp	Salary (USD)	End revision timestamp
2008	MOD	2010-02-01	4000	null
2009	ADD	2010-04-01	4500	null

15.12.3.1. Determining a suitable partitioning column

To partition this data, the 'level of interestingness' 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 interesting because it is 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.

2. 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 uninteresting. The only interesting 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.

15.12.3.2. Determining a suitable partitioning scheme

A possible partitioning scheme for the salary table would be as follows:

1. *end revision timestamp* year = 2008

   This partition contains audit data that is not very (or no longer) interesting.

2. *end revision timestamp* year = 2009

This partition contains audit data that is potentially interesting.

3. *end revision timestamp* year >= 2010 or null

   This partition contains the most interesting 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 interesting (in 2011).

2. *end revision timestamp* year >= 2011 or null

   This partition contains the most interesting audit data and is the new 'extension bucket'.

15.13. Envers links

1. Hibernate main page
2. Forum
3. JIRA issue tracker (when adding issues concerning Envers, be sure to select the "envers" component!)
4. IRC channel
5. Envers Blog
6. FAQ

Chapter 16. Multi-tenancy

Table of Contents

16.1. What is multi-tenancy?

The term multi-tenancy 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 multi-tenant data, on which we will focus.

16.2. Multi-tenant data approaches

There are 3 main approaches to isolating information in these multi-tenant systems which goes hand-in-hand with different database schema definitions and JDBC setups.

Note

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. One example is http://msdn.microsoft.com/en-us/library/aa479086.aspx which does a great job of covering these topics.

16.2.1. 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.

16.2.2. Separate schema

Each tenant's data is kept in a distinct database schema on a single database instance. There are 2 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.

16.2.3. 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.

16.3. Multi-tenancy in Hibernate

Using Hibernate with multi-tenant 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 16.1. Specifying tenant identifier from `SessionFactory`

```
Session session = sessionFactory.withOptions()
        .tenantIdentifier( yourTenantIdentifier )
        ...
        .openSession();
```

Additionally, when specifying configuration, a `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 multi-tenancy 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 `org.hibernate.service.jdbc.connections.spi.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 `org.hibernate.service.jdbc.connections.spi.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 in Hibernate as of 4.0 and 4.1. Its support is planned for 5.0.

16.3.1. `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 `org.hibernate.service.jdbc.connections.spi.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.service.ServiceRegistryBuilder`.
- 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 `org.hibernate.service.jdbc.connections.spi.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 for more details.

16.3.2. `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 2 situations where `CurrentTenantIdentifierResolver` is used:

- The first situation is when the application is using the `org.hibernate.context.spi.CurrentSessionContext` feature in conjunction with multi-tenancy. 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 multi-tenant 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` be supplied.
- The other situation is when you do not want to have to explicitly specify the tenant identifier all the time as we saw in Example 16.1, "Specifying tenant identifier from `SessionFactory`". 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.

16.3.3. Caching

Multi-tenancy support in Hibernate works seamlessly with the Hibernate second level cache. The key used to cache data encodes the tenant identifier.

16.3.4. Odds and ends

Currently schema export will not really work with multi-tenancy. That may not change.

The JPA expert group is in the process of defining multi-tenancy support for the upcoming 2.1 version of the specification.

16.4. Strategies for `MultiTenantConnectionProvider` implementors

Example 16.2. Implementing MultiTenantConnectionProvider using different connection pools

```
/**
 * Simplisitc implementation for illustration purposes supporting 2 hard coded providers (pools) and leveraging
 * the support class {@link org.hibernate.service.jdbc.connections.spi.AbstractMultiTenantConnectionProvider}
 */
public class MultiTenantConnectionProviderImpl extends AbstractMultiTenantConnectionProvider {
        private final ConnectionProvider acmeProvider = ConnectionProviderUtils.buildConnectionProvider( "acme" );
        private final ConnectionProvider jbossProvider = ConnectionProviderUtils.buildConnectionProvider( "jboss" );

        @Override
        protected ConnectionProvider getAnyConnectionProvider() {
                return acmeProvider;
        }

        @Override
        protected ConnectionProvider selectConnectionProvider(String tenantIdentifier) {
```

```
		if ( "acme".equals( tenantIdentifier ) ) {
			return acmeProvider;
		}
		else if ( "jboss".equals( tenantIdentifier ) ) {
			return jbossProvider;
		}
		throw new HibernateException( "Unknown tenant identifier" );
	}
}
```

The approach above is valid for the DATABASE approach. It is also valid for the SCHEMA approach provided the underlying database allows naming the schema to which to connect in the connection URL.

Example 16.3. Implementing MultiTenantConnectionProvider using single connection pool

```
/**
 * Simplisitc implementation for illustration purposes showing a single connection pool used to serve
 * multiple schemas using "connection altering".  Here we use the T-SQL specific USE command; Oracle
 * users might use the ALTER SESSION SET SCHEMA command; etc.
 */
public class MultiTenantConnectionProviderImpl
		implements MultiTenantConnectionProvider, Stoppable {
	private final ConnectionProvider connectionProvider = ConnectionProviderUtils.buildConnectionProvider( "master" );

	@Override
	public Connection getAnyConnection() throws SQLException {
		return connectionProvider.getConnection();
	}

	@Override
	public void releaseAnyConnection(Connection connection) throws SQLException {
		connectionProvider.closeConnection( connection );
	}

	@Override
	public Connection getConnection(String tenantIdentifier) throws SQLException {
		final Connection connection = getAnyConnection();
```

```
                try {
                        connection.createStatement().execute( "USE " + tenanantIdentifier );
                }
                catch ( SQLException e ) {
                        throw new HibernateException(
                                        "Could not alter JDBC connection to specified schema [" +
                                                        tenantIdentifier + "]",
                                        e
                        );
                }
                return connection;
        }

        @Override
        public void releaseConnection(String tenantIdentifier, Connection connection) throws SQLException {
                try {
                        connection.createStatement().execute( "USE master" );
                }
                catch ( SQLException e ) {
                        // on error, throw an exception to make sure the connection is not returned to the pool.
                        // your requirements may differ
                        throw new HibernateException(
                                        "Could not alter JDBC connection to specified schema [" +
                                                        tenantIdentifier + "]",
                                        e
                        );
                }
                connectionProvider.closeConnection( connection );
        }

        ...
}
```

This approach is only relevant to the SCHEMA approach.

Appendix A. Configuration properties

Table of Contents

A.1. General Configuration

hibernate.dialect	A fully-qualified classname	The classname of a Hibernate `org.hibernate.dialect.Dialect` from which Hibernate can generate SQL optimized for a particular relational database. In most cases Hibernate can choose the correct `org.hibernate.dialect.Dialect` implementation based on the JDBC metadata returned by the JDBC driver.
hibernate.show_sql	`true` or `false`	Write all SQL statements to the console. This is an alternative to setting the log category org.hibernate.SQL to debug.
hibernate.format_sql	`true` or `false`	Pretty-print the SQL in the log and console.
hibernate.default_schema	A schema name	Qualify unqualified table names with the given schema or tablespace in generated SQL.
hibernate.default_catalog	A catalog name	Qualifies unqualified table names with the given catalog in generated SQL.
hibernate.session_factory_name	A JNDI name	The `org.hibernate.SessionFactory` is automatically bound to this name in JNDI after it is created.
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 assocation 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.
hibernate.default_entity_mode	`dynamic-map` or `pojo`	Default mode for entity representation for all sessions opened from this `SessionFactory`, defaults to `pojo`.
hibernate.order_updates	`true` or `false`	Forces Hibernate to order SQL updates by the primary key value of the items being updated. This reduces the likelihood of transaction deadlocks in highly-concurrent systems.
hibernate.generate_statistics	`true` or `false`	Causes Hibernate to collect statistics for performance tuning.
hibernate.use_identifier_rollback	`true` or `false`	If true, generated identifier properties are reset to default values when objects are deleted.
hibernate.use_sql_comments	`true` or `false`	If true, Hibernate generates comments inside the SQL, for easier debugging.

A.2. Database configuration

Table A.1. JDBC properties

Property	Example	Purpose
hibernate.jdbc.fetch_size	`0` or an integer	A non-zero value determines the JDBC fetch size, by calling `Statement.setFetchSize()`.
hibernate.jdbc.batch_size	A value between `5` and `30`	A non-zero value causes Hibernate to use JDBC2 batch updates.
hibernate.jdbc.batch_versioned_data	`true` or `false`	Set this property to `true` if your JDBC driver returns correct row counts from `executeBatch()`. This option is usually safe, but is disabled by default. If enabled, Hibernate uses batched DML for automatically versioned data.
hibernate.jdbc.factory_class	The fully-qualified class name of the factory	Select a custom `org.hibernate.jdbc.Batcher`. Irrelevant for most applications.
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`	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.

Table A.2. Cache Properties

Property	Example	Purpose
hibernate.cache.provider_class	Fully-qualified classname	The classname of a custom CacheProvider.

Property	Example	Purpose
hibernate.cache.use_minimal_puts	`true` or `false`	Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This is most useful for clustered caches and is enabled by default for clustered cache implementations.
hibernate.cache.use_query_cache	`true` or `false`	Enables the query cache. You still need to set individual queries to be cachable.
hibernate.cache.use_second_level_cache	`true` or `false`	Completely disable the second level cache, which is enabled by default for classes which specify a <cache> mapping.
hibernate.cache.query_cache_factory	Fully-qualified classname	A custom `QueryCache` interface. The default is the built-in `StandardQueryCache`.
hibernate.cache.region_prefix	A string	A prefix for second-level cache region names.
hibernate.cache.use_structured_entries	`true` or `false`	Forces Hibernate to store data in the second-level cache in a more human-readable format.

Table A.3. Transactions properties

Property	Example	Purpose
hibernate.transaction.factory_class	A fully-qualified classname	The classname of a `TransactionFactory` to use with Hibernate Transaction API. The default is `JDBCTransactionFactory`).
jta.UserTransaction	A JNDI name	The `JTATransactionFactory` needs a JNDI name to obtain the JTA UserTransaction from the application server.
hibernate.transaction.manager_lookup_class	A fully-qualified classname	The classname of a `TransactionManagerLookup`, which is used in conjunction with JVM-level or the hilo generator in a JTA environment.
hibernate.transaction.flush_before_completion	`true` or `false`	Causes the session be flushed during the before completion phase of the transaction. If possible, use built-in and automatic session context management instead.
hibernate.transaction.auto_close_session	`true` or `false`	Causes the session to be closed during the after completion phase of the transaction. If possible, use built-in and automatic session context management instead.

Note

Each of the properties in the following table are prefixed by `hibernate.`. It has been removed in the table to conserve space.

Table A.4. Miscellaneous properties

Property	Example	Purpose
current_session_context_class	One of `jta`, `thread`, `managed`, or `custom.Class`	Supply a custom strategy for the scoping of the `Current` Session.
factory_class	`org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory` or `org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory`	Chooses the HQL parser implementation.
query.substitutions	`hqlLiteral=SQL_LITERAL` or `hqlFunction=SQLFUNC`	Map from tokens in Hibernate queries to SQL tokens, such as function or literal names.
hbm2ddl.auto	`validate`, `update`, `create`, `create-drop`	Validates or exports schema DDL to the database when the `SessionFactory` is created. With **create-drop**, the database schema is dropped when the `SessionFactory` is closed explicitly.
cglib.use_reflection_optimizer	`true` or `false`	If enabled, Hibernate uses CGLIB instead of runtime reflection. This is a system-level property. Reflection is useful for troubleshooting. Hibernate always requires CGLIB even if you disable the optimizer. You cannot set this property in hibernate.cfg.xml.

A.3. Connection pool properties

c3p0 connection pool properties

- hibernate.c3p0.min_size
- hibernate.c3p0.max_size
- hibernate.c3p0.timeout

- hibernate.c3p0.max_statements

Table A.5. Proxool connection pool properties

Property	Description
hibernate.proxool.xml	Configure Proxool provider using an XML file (.xml is appended automatically)
hibernate.proxool.properties	Configure the Proxool provider using a properties file (.properties is appended automatically)
hibernate.proxool.existing_pool	Whether to configure the Proxool provider from an existing pool
hibernate.proxool.pool_alias	Proxool pool alias to use. Required.

Note

For information on specific configuration of Proxool, refer to the Proxool documentation available from http://proxool.sourceforge.net/.

Appendix B. Legacy Hibernate Criteria Queries

Table of Contents

Note

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 ???.

This information is copied as-is from the older Hibernate documentation.

Hibernate features an intuitive, extensible criteria query API.

B.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.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

B.2. 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.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Restrictions can be grouped logically.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

There are a range of built-in criterion types (`Restrictions` subclasses). One of the most useful allows you to specify SQL directly.

```
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()`:

```
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();
```

B.3. Ordering the results

You can order the results using `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

B.4. Associations

By navigating associations using `createCriteria()` you can specify constraints upon related entities:

```
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:

```
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`.

```
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 retreive 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.

B.5. Dynamic association fetching

You can specify association fetching semantics at runtime using `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join. See ??? for more information.

B.6. 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 Section B.7, “Collections”

B.7. 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 Section B.4, “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 queryng 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();
```

B.8. Example queries

The class `org.hibernate.criterion.Example` allows you to construct a query criterion from a given instance.

```
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.

```
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.

```
List results = session.createCriteria(Cat.class)
```

```
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

B.9. 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()`.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

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:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
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:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

You can also use `Property.forName()` to express projections:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

B.10. 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`.

```
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`.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight").gt(avgWeight) )
    .list();
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Correlated subqueries are also possible:

```
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:

```
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();
```

B.11. 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.

```
<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.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```
