Chapter 3. Configuration

Hibernate is designed to operate in many different environments and, as such, there is a broad range of configuration parameters. Fortunately, most have sensible default values and Hibernate is distributed with an example hibernate.properties file in etc/ that displays the various options. Simply put the example file in your classpath and customize it to suit your needs.

3.1. Programmatic configuration

An instance of 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 is used to build an immutable org.hibernate.SessionFactory. The mappings are compiled from various XML mapping files.

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 addResource(). For example:

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

An alternative way is to specify the mapped class and allow Hibernate to find the mapping document for you:

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

Hibernate will then search for mapping files named **/org/hibernate/auction/Item.hbm.xml** and **/org/hibernate/auction/Bid.hbm.xml** in the classpath. This approach eliminates any hardcoded filenames.

A org.hibernate.cfg.Configuration also allows you to specify configuration properties. For example:

```
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");
```

This is not the only way to pass configuration properties to Hibernate. Some alternative options include:

1. Pass an instance of java.util.Properties to Configuration.setProperties().
2. Place a file named **hibernate.properties** in a root directory of the classpath.
3. Set System properties using java -Dproperty=value.
4. Include <property> elements in hibernate.cfg.xml (this is discussed later).

If you want to get started quickly**hibernate.properties** is the easiest approach.

The org.hibernate.cfg.Configuration is intended as a startup-time object that will be discarded once a SessionFactory is created.

3.2. Obtaining a SessionFactory

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

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

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

3.3. JDBC connections

It is advisable to have the org.hibernate.SessionFactory create and pool JDBC connections for you. If you take this approach, opening a org.hibernate.Session is as simple as:

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

Once you start a task that requires access to the database, a JDBC connection will be obtained from the pool.

Before you can do this, you first need to pass some JDBC connection properties to Hibernate. All Hibernate property names and semantics are defined on the class org.hibernate.cfg.Environment. The most important settings for JDBC connection configuration are outlined below.

Hibernate will obtain and pool connections using java.sql.DriverManager if you set the following properties:

Table 3.1. Hibernate JDBC Properties

Property name	Purpose
hibernate.connection.driver_class	*JDBC driver class*
hibernate.connection.url	*JDBC URL*
hibernate.connection.username	*database user*
hibernate.connection.password	*database user password*
hibernate.connection.pool_size	*maximum number of pooled connections*

Hibernate's own connection pooling algorithm is, however, quite rudimentary. It is intended to help you get started and is *not intended for use in a production system*, or even for performance testing. You should use a third party pool for best performance and stability. Just replace the hibernate.connection.pool_size property with connection pool specific settings. This will turn off Hibernate's internal pool. For example, you might like to use c3p0.

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the **lib** directory. Hibernate will use its org.hibernate.connection.C3P0ConnectionProvider for connection pooling if you set hibernate.c3p0.* properties. If you would like to use Proxool, refer to the packaged **hibernate.properties** and the Hibernate web site for more information.

The following is an example **hibernate.properties** file for c3p0:

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

For use inside an application server, you should almost always configure Hibernate to obtain connections from an application server javax.sql.Datasource registered in JNDI. You will need to set at least one of the following properties:

Table 3.2. Hibernate Datasource Properties

Property name	Purpose
hibernate.connection.datasource	*datasource JNDI name*
hibernate.jndi.url	*URL of the JNDI provider* (optional)
hibernate.jndi.class	*class of the JNDI InitialContextFactory* (optional)
hibernate.connection.username	*database user* (optional)
hibernate.connection.password	*database user password* (optional)

Here is an example **hibernate.properties** file for an application server provided JNDI datasource:

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

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

Arbitrary connection properties can be given by prepending "hibernate.connection" to the connection property name. For example, you can specify a charSet connection property using hibernate.connection.charSet.

You can define your own plugin strategy for obtaining JDBC connections by implementing the interface org.hibernate.connection.ConnectionProvider, and specifying your custom implementation via the hibernate.connection.provider_class property.

3.4. Optional configuration properties

There are a number of other properties that control the behavior of Hibernate at runtime. All are optional and have reasonable default values.

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

Table 3.3. Hibernate Configuration Properties

Property name	Purpose
hibernate.dialect	The classname of a Hibernate org.hibernate.dialect.Dialect which allows Hibernate to generate SQL optimized for a particular relational database. **e.g.** full.classname.of.Dialect In most cases Hibernate will actually be able to choose the correct org.hibernate.dialect.Dialect implementation based on the JDBC metadata returned by the JDBC driver.
hibernate.show_sql	Write all SQL statements to console. This is an alternative to setting the log category org.hibernate.SQL to debug. **e.g.** true | false
hibernate.format_sql	Pretty print the SQL in the log and console. **e.g.** true | false
hibernate.default_schema	Qualify unqualified table names with the given schema/tablespace in generated SQL. **e.g.** SCHEMA_NAME
hibernate.default_catalog	Qualifies unqualified table names with the given catalog in generated SQL. **e.g.** CATALOG_NAME
hibernate.session_factory_name	The org.hibernate.SessionFactory will be automatically bound to this name in JNDI after it has been created. **e.g.** jndi/composite/name
hibernate.max_fetch_depth	Sets a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0 disables default outer join fetching. **e.g.** recommended values between 0 and 3
hibernate.default_batch_fetch_size	Sets a default size for Hibernate batch fetching of associations. **e.g.** recommended values 4, 8, 16
hibernate.default_entity_mode	Sets a default mode for entity representation for all sessions opened from this SessionFactory dynamic-map, dom4j, pojo
hibernate.order_updates	Forces Hibernate to order SQL updates by the primary key value of the items being updated. This will result in fewer transaction deadlocks in highly concurrent systems. **e.g.** true | false
hibernate.generate_statistics	If enabled, Hibernate will collect statistics useful for performance tuning. **e.g.** true | false
hibernate.use_identifier_rollback	If enabled, generated identifier properties will be reset to default values when objects are deleted. **e.g.** true | false

Property name	Purpose
hibernate.use_sql_comments	If turned on, Hibernate will generate comments inside the SQL, for easier debugging, defaults to false. **e.g.** true | false

Table 3.4. Hibernate JDBC and Connection Properties

Property name	Purpose
hibernate.jdbc.fetch_size	A non-zero value determines the JDBC fetch size (calls Statement.setFetchSize()).
hibernate.jdbc.batch_size	A non-zero value enables use of JDBC2 batch updates by Hibernate. **e.g.** recommended values between 5 and 30
hibernate.jdbc.batch_versioned_data	Set this property to true if your JDBC driver returns correct row counts from executeBatch(). Iit is usually safe to turn this option on. Hibernate will then use batched DML for automatically versioned data. Defaults to false. **e.g.** true | false
hibernate.jdbc.factory_class	Select a custom org.hibernate.jdbc.Batcher. Most applications will not need this configuration property. **e.g.** classname.of.BatcherFactory
hibernate.jdbc.use_scrollable_resultset	Enables use of JDBC2 scrollable resultsets by Hibernate. This property is only necessary when using user-supplied JDBC connections. Hibernate uses connection metadata otherwise. **e.g.** true | false
hibernate.jdbc.use_streams_for_binary	Use streams when writing/reading binary or serializable types to/from JDBC. **system-level property** **e.g.** true | false
hibernate.jdbc.use_get_generated_keys	Enables use of JDBC3 PreparedStatement.getGeneratedKeys() to retrieve natively generated keys after insert. Requires JDBC3+ driver and JRE1.4+, set to false if your driver has problems with the Hibernate identifier generators. By default, it tries to determine the driver capabilities using connection metadata. **e.g.** true|false
hibernate.connection.provider_class	The classname of a custom org.hibernate.connection.ConnectionProvider which provides JDBC connections to Hibernate. **e.g.** classname.of.ConnectionProvider
hibernate.connection.isolation	Sets the JDBC transaction isolation level. Check java.sql.Connection for meaningful values, but note that most databases do not support all isolation levels and some define additional, non-standard isolations. **e.g.** 1, 2, 4, 8
hibernate.connection.autocommit	Enables autocommit for JDBC pooled connections (it is not recommended). **e.g.** true | false

Property name	Purpose
hibernate.connection.release_mode	Specifies when Hibernate should release JDBC connections. By default, a JDBC connection is held until the session is explicitly closed or disconnected. For an application server JTA datasource, use after_statement to aggressively release connections after every JDBC call. For a non-JTA connection, it often makes sense to release the connection at the end of each transaction, by using after_transaction. auto will choose after_statement for the JTA and CMT transaction strategies and after_transaction for the JDBC transaction strategy. **e.g.** auto (default) | on_close | after_transaction | after_statement This setting only affects Sessions returned from SessionFactory.openSession. For Sessions obtained through SessionFactory.getCurrentSession, the CurrentSessionContext implementation configured for use controls the connection release mode for those Sessions. See Section 2.5, "Contextual sessions"
hibernate.connection.*<propertyName>*	Pass the JDBC property *<propertyName>* to DriverManager.getConnection().
hibernate.jndi.*<propertyName>*	Pass the property *<propertyName>* to the JNDI InitialContextFactory.

Table 3.5. Hibernate Cache Properties

Property name	Purpose
hibernate.cache.provider_class	The classname of a custom CacheProvider. **e.g.** classname.of.CacheProvider
hibernate.cache.use_minimal_puts	Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate3, is enabled by default for clustered cache implementations. **e.g.** true|false
hibernate.cache.use_query_cache	Enables the query cache. Individual queries still have to be set cachable. **e.g.** true|false
hibernate.cache.use_second_level_cache	Can be used to completely disable the second level cache, which is enabled by default for classes which specify a <cache> mapping. **e.g.** true|false
hibernate.cache.query_cache_factory	The classname of a custom QueryCache interface, defaults to the built-in StandardQueryCache. **e.g.** classname.of.QueryCache
hibernate.cache.region_prefix	A prefix to use for second-level cache region names. **e.g.** prefix
hibernate.cache.use_structured_entries	Forces Hibernate to store data in the second-level cache in a more human-friendly format. **e.g.** true|false

Table 3.6. Hibernate Transaction Properties

Property name	Purpose

Property name	Purpose
hibernate.transaction.factory_class	The classname of a TransactionFactory to use with Hibernate Transaction API (defaults to JDBCTransactionFactory). **e.g.** classname.of.TransactionFactory
jta.UserTransaction	A JNDI name used by JTATransactionFactory to obtain the JTA UserTransaction from the application server. **e.g.** jndi/composite/name
hibernate.transaction.manager_lookup_class	The classname of a TransactionManagerLookup. It is required when JVM-level caching is enabled or when using hilo generator in a JTA environment. **e.g.** classname.of.TransactionManagerLookup
hibernate.transaction.flush_before_completion	If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see Section 2.5, “Contextual sessions”. **e.g.** true | false
hibernate.transaction.auto_close_session	If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see Section 2.5, “Contextual sessions”. **e.g.** true | false

Table 3.7. Miscellaneous Properties

Property name	Purpose
hibernate.current_session_context_class	Supply a custom strategy for the scoping of the "current" Session. See Section 2.5, “Contextual sessions” for more information about the built-in strategies. **e.g.** jta | thread | managed | custom.Class
hibernate.query.factory_class	Chooses the HQL parser implementation. **e.g.** org.hibernate.hql.ast.ASTQueryTranslatorFactory or org.hibernate.hql.classic.ClassicQueryTranslatorFactory
hibernate.query.substitutions	Is used to map from tokens in Hibernate queries to SQL tokens (tokens might be function or literal names, for example). **e.g.** hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC
hibernate.hbm2ddl.auto	Automatically validates or exports schema DDL to the database when the SessionFactory is created. With create-drop, the database schema will be dropped when the SessionFactory is closed explicitly. **e.g.** validate | update | create | create-drop
hibernate.cglib.use_reflection_optimizer	Enables the use of CGLIB instead of runtime reflection (System-level property). Reflection can sometimes be useful when troubleshooting. Hibernate always requires CGLIB even if you turn off the optimizer. You cannot set this property in hibernate.cfg.xml. **e.g.** true | false

3.4.1. SQL Dialects

Always set the `hibernate.dialect` property to the correct `org.hibernate.dialect.Dialect` subclass for your database. If you specify a dialect, Hibernate will use sensible defaults for some of the other properties listed above. This means that you will not have to specify them manually.

Table 3.8. Hibernate SQL Dialects (`hibernate.dialect`)

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

3.4.2. Outer Join Fetching

If your database supports ANSI, Oracle or Sybase style outer joins, *outer join fetching* will often increase performance by limiting the number of round trips to and from the database. This is, however, at the cost of possibly more work performed by the database itself. Outer join fetching allows a whole graph of objects connected by many-to-one, one-to-many, many-to-many and one-to-one associations to be retrieved in a single SQL `SELECT`.

Outer join fetching can be disabled *globally* by setting the property `hibernate.max_fetch_depth` to `0`. A setting of `1` or higher enables outer join fetching for one-to-one and many-to-one associations that have been mapped with `fetch="join"`.

See Section 19.1, “Fetching strategies” for more information.

3.4.3. Binary Streams

Oracle limits the size of `byte` arrays that can be passed to and/or from its JDBC driver. If you wish to use large instances of `binary` or `serializable` type, you should enable `hibernate.jdbc.use_streams_for_binary`. *This is a system-level setting only.*

3.4.4. Second-level and query cache

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the Section 19.2, “The Second Level Cache” for more information.

3.4.5. Query Language Substitution

You can define new Hibernate query tokens using hibernate.query.substitutions. For example:

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

This would cause the tokens true and false to be translated to integer literals in the generated SQL.

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

This would allow you to rename the SQL LOWER function.

3.4.6. Hibernate statistics

If you enable hibernate.generate_statistics, Hibernate exposes a number of metrics that are useful when tuning a running system via SessionFactory.getStatistics(). Hibernate can even be configured to expose these statistics via JMX. Read the Javadoc of the interfaces in org.hibernate.stats for more information.

3.5. Logging

Hibernate utilizes Simple Logging Facade for Java (SLF4J) in order to log various system events. SLF4J can direct your logging output to several logging frameworks (NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL or logback) depending on your chosen binding. In order to setup logging you will need **slf4j-api.jar** in your classpath together with the jar file for your preferred binding - **slf4j-log4j12.jar** in the case of Log4J. See the SLF4J documentation for more detail. To use Log4j you will also need to place a **log4j.properties** file in your classpath. An example properties file is distributed with Hibernate in the src/ directory.

It is recommended that you familiarize yourself with Hibernate's log messages. A lot of work has been put into making the Hibernate log as detailed as possible, without making it unreadable. It is an essential troubleshooting device. The most interesting log categories are the following:

Table 3.9. Hibernate Log Categories

Category	Function
org.hibernate.SQL	Log all SQL DML statements as they are executed
org.hibernate.type	Log all JDBC parameters
org.hibernate.tool.hbm2ddl	Log all SQL DDL statements as they are executed
org.hibernate.pretty	Log the state of all entities (max 20 entities) associated with the session at flush time
org.hibernate.cache	Log all second-level cache activity
org.hibernate.transaction	Log transaction related activity
org.hibernate.jdbc	Log all JDBC resource acquisition
org.hibernate.hql.ast.AST	Log HQL and SQL ASTs during query parsing
org.hibernate.secure	Log all JAAS authorization requests
org.hibernate	Log everything. This is a lot of information but it is useful for troubleshooting

When developing applications with Hibernate, you should almost always work with debug enabled for the category org.hibernate.SQL, or, alternatively, the property hibernate.show_sql enabled.

3.6. Implementing a NamingStrategy

The interface org.hibernate.cfg.NamingStrategy allows you to specify a "naming standard" for database objects and schema elements.

You can provide rules for automatically generating database identifiers from Java identifiers or for processing "logical" column and table names given in the mapping file into "physical" table and column names. This feature helps reduce the verbosity of the mapping document, eliminating repetitive noise (TBL_ prefixes, for example). The default strategy used by Hibernate is quite minimal.

You can specify a different strategy by calling Configuration.setNamingStrategy() before adding mappings:

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

org.hibernate.cfg.ImprovedNamingStrategy is a built-in strategy that might be a useful starting point for some applications.

3.7. XML configuration file

An alternative approach to configuration is to specify a full configuration in a file named hibernate.cfg.xml. This file can be used as a replacement for the hibernate.properties file or, if both are present, to override properties.

The XML configuration file is by default expected to be in the root of your CLASSPATH. Here is an example:

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

<hibernate-configuration>

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

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

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

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

    </session-factory>

</hibernate-configuration>
```

The advantage of this approach is the externalization of the mapping file names to configuration. The hibernate.cfg.xml is also more convenient once you have to tune the Hibernate cache. It is your choice to use either hibernate.properties or hibernate.cfg.xml. Both are equivalent, except for the above mentioned benefits of using the XML syntax.

With the XML configuration, starting Hibernate is then as simple as:

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

You can select a different XML configuration file using:

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

3.8. J2EE Application Server integration

Hibernate has the following integration points for J2EE infrastructure:

- *Container-managed datasources*: Hibernate can use JDBC connections managed by the container and provided through JNDI. Usually, a JTA compatible TransactionManager and a ResourceManager take care of transaction management (CMT), especially distributed transaction handling across several datasources. You can also demarcate transaction boundaries programmatically (BMT), or you might want to use the optional Hibernate Transaction API for this to keep your code portable.

- *Automatic JNDI binding*: Hibernate can bind its SessionFactory to JNDI after startup.

- *JTA Session binding:* the Hibernate Session can be automatically bound to the scope of JTA transactions. Simply lookup the SessionFactory from JNDI and get the current Session. Let Hibernate manage flushing and closing the Session when your JTA transaction completes. Transaction demarcation is either declarative (CMT) or programmatic (BMT/UserTransaction).

- *JMX deployment:* if you have a JMX capable application server (e.g. JBoss AS), you can choose to deploy Hibernate as a managed MBean. This saves you the one line startup code to build your SessionFactory from a Configuration. The container will startup your HibernateService and also take care of service dependencies (datasource has to be available before Hibernate starts, etc).

Depending on your environment, you might have to set the configuration option hibernate.connection.aggressive_release to true if your application server shows "connection containment" exceptions.

3.8.1. Transaction strategy configuration

The Hibernate Session API is independent of any transaction demarcation system in your architecture. If you let Hibernate use JDBC directly through a connection pool, you can begin and end your transactions by calling the JDBC API. If you run in a J2EE application server, you might want to use bean-managed transactions and call the JTA API and UserTransaction when needed.

To keep your code portable between these two (and other) environments we recommend the optional Hibernate Transaction API, which wraps and hides the underlying system. You have to specify a factory class for Transaction instances by setting the Hibernate configuration property hibernate.transaction.factory_class.

There are three standard, or built-in, choices:

org.hibernate.transaction.JDBCTransactionFactory

delegates to database (JDBC) transactions (default)

org.hibernate.transaction.JTATransactionFactory

delegates to container-managed transactions if an existing transaction is underway in this context (for example, EJB session bean method). Otherwise, a new transaction is started and bean-managed transactions are used.

org.hibernate.transaction.CMTTransactionFactory

delegates to container-managed JTA transactions

You can also define your own transaction strategies (for a CORBA transaction service, for example).

Some features in Hibernate (i.e., the second level cache, Contextual Sessions with JTA, etc.) require access to the JTA TransactionManager in a managed environment. In an application server, since J2EE does not standardize a single mechanism, you have to specify how Hibernate should obtain a reference to the TransactionManager:

Table 3.10. JTA TransactionManagers

Transaction Factory	Application Server
org.hibernate.transaction.JBossTransactionManagerLookup	JBoss
org.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
org.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
org.hibernate.transaction.WebSphereExtendedJTATransactionLookup	WebSphere 6
org.hibernate.transaction.OrionTransactionManagerLookup	Orion
org.hibernate.transaction.ResinTransactionManagerLookup	Resin
org.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
org.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS
org.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
org.hibernate.transaction.BESTransactionManagerLookup	Borland ES

3.8.2. JNDI-bound SessionFactory

A JNDI-bound Hibernate SessionFactory can simplify the lookup function of the factory and create new Sessions. This is not, however, related to a JNDI bound Datasource; both simply use the same registry.

If you wish to have the SessionFactory bound to a JNDI namespace, specify a name (e.g. java:hibernate/SessionFactory) using the property hibernate.session_factory_name. If this property is omitted, the SessionFactory will not be bound to JNDI. This is especially useful in environments with a read-only JNDI default implementation (in Tomcat, for example).

When binding the SessionFactory to JNDI, Hibernate will use the values of hibernate.jndi.url, hibernate.jndi.class to instantiate an initial context. If they are not specified, the default InitialContext will be used.

Hibernate will automatically place the SessionFactory in JNDI after you call cfg.buildSessionFactory(). This means you will have this call in some startup code, or utility class in your application, unless you use JMX deployment with the HibernateService (this is discussed later in greater detail).

If you use a JNDI SessionFactory, an EJB or any other class, you can obtain the SessionFactory using a JNDI lookup.

It is recommended that you bind the SessionFactory to JNDI in a managed environment and use a static singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a SessionFactory in a helper class, such as HibernateUtil.getSessionFactory(). Note that such a class is also a convenient way to startup Hibernatesee chapter 1.

3.8.3. Current Session context management with JTA

The easiest way to handle Sessions and transactions is Hibernate's automatic "current" Session management. For a discussion of contextual sessions see Section 2.5, “Contextual sessions”. Using the "jta" session context, if there is no Hibernate Session associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call sessionFactory.getCurrentSession(). The Sessions retrieved via getCurrentSession() in the"jta" context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the Sessions to be managed by the life cycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through UserTransaction, or (recommended for portable code) use the Hibernate Transaction API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

3.8.4. JMX deployment

The line cfg.buildSessionFactory() still has to be executed somewhere to get a SessionFactory into JNDI. You can do this either in a static initializer block, like the one in HibernateUtil, or you can deploy Hibernate as a *managed service*.

Hibernate is distributed with org.hibernate.jmx.HibernateService for deployment on an application server with JMX capabilities, such as JBoss AS. The actual deployment and configuration is vendor-specific. Here is an example jboss-service.xml for JBoss 4.0.x:

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

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

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

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

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

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

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

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

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

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

</mbean>

</server>
```

This file is deployed in a directory called META-INF and packaged in a JAR file with the extension .sar (service archive). You also need to package Hibernate, its required third-party libraries, your compiled persistent classes, as well as your mapping files in the same archive. Your enterprise beans (usually session beans) can be kept in their own JAR file, but you can include this EJB JAR file in the main service archive to get a single (hot-)deployable unit. Consult the JBoss AS documentation for more information about JMX service and EJB deployment.

Prev Top of page Front page Next

Chapter 2. Architecture Chapter 4. Persistent Classes