Hibernate Configuration

As Hibernate can operate in different environments, it requires a wide range of configuration parameters. These configurations contain the mapping information that provides different functionalities to Java classes.

Generally, we provide database related mappings in the configuration file.

Hibernate facilitates to provide the configurations either in an XML file (like **hibernate.cfg.xml)** or properties file (like **hibernate.properties**).

We can configure Hibernate in three ways: -

1. **Programmatic configuration:** Use the API to load the **hbm** file, load the database driver, and specify the database connection details.
2. **XML configuration:** Specify the database connection details in an XML file that's loaded along with the **hbm** file. The default file name is **hibernate.cfg.xml.** You can use another name by specifying the name explicitly.
3. **Properties file configuration:** Similar to the XML configuration, but uses a **.properties** file. The default name is **hibernate.properties**.

Programmatic Configuration

The following code loads the configuration programmatically. If you have a very specific use case to configure programmatically, you can use this method; otherwise, the preferred way is to use **annotations**.

The Configuration class provides the API to load the hbm files, to specify the driver to be used for the database connection, and to provide other connection details:

```
Configuration configuration = new Configuration()
.addResource("com/metaarchit/bookshop/Book.hbm.xml")
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyTenSevenDialect")
.setProperty("hibernate.connection.driver_class",
"org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url",
"jdbc:derby://localhost:1527/BookShopDB")
```

```
.setProperty("hibernate.connection.username", "book")
.setProperty("hibernate.connection.password", "book");
ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Instead of using *addResource*() to add the mapping files, you can also use *addClass()* to add a persistent class and let Hibernate load the mapping definition for this class:

```
Configuration configuration = new Configuration()
.addClass(com.metaarchit.bookshop.Book.class)
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyDialect")
.setProperty("hibernate.connection.driver_class",
"org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url",
"jdbc:derby://localhost:1527/BookShopDB")
.setProperty("hibernate.connection.username", "book")
.setProperty("hibernate.connection.password", "book");
ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

If your application has hundreds of mapping definitions, you can pack it in a JAR file and add it to the Hibernate configuration. This JAR file must be found in your application's classpath:

```
Configuration configuration = new Configuration()
.addJar(new File("mapping.jar"))
.setProperty("hibernate.dialect", "org.hibernate.dialect.DerbyDialect")
.setProperty("hibernate.connection.driver_class",
"org.apache.derby.jdbc.EmbeddedDriver")
.setProperty("hibernate.connection.url",
"jdbc:derby://localhost:1527/BookShopDB")
.setProperty("hibernate.connection.username", "book")
```

```
.setProperty("hibernate.connection.password", "book");
ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder().applySettings
(configuration.getProperties()).build();
sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Properties of Hibernate Configuration

Hibernate JDBC Properties

Property	Description
hibernate.connection.driver_class	It represents the JDBC driver class.
hibernate.connection.url	It represents the JDBC URL.
hibernate.connection.username	It represents the database username.
hibernate.connection.password	It represents the database password.
Hibernate.connection.pool_size	It represents the maximum number of connections available in the connection pool.

Hibernate Datasource Properties

Property	Description
hibernate.connection.datasource	It represents datasource JNDI name which is used by Hibernate for database properties.
hibernate.jndi.url	It is optional. It represents the URL of the JNDI provider.
hibernate.jndi.class	It is optional. It represents the class of the JNDI InitialContextFactory.

Hibernate Configuration Properties

Property	Description
hibernate.dialect	It represents the type of database used in hibernate to generate SQL statements for a particular relational database.
hibernate.show_sql	It is used to display the executed SQL statements to console.

hibernate.format_sql	It is used to print the SQL in the log and console.
hibernate.default_catalog	It qualifies unqualified table names with the given catalog in generated SQL.
hibernate.default_schema	It qualifies unqualified table names with the given schema in generated SQL.
hibernate.session_factory_name	The SessionFactory interface automatically bound to this name in JNDI after it has been created.
hibernate.default_entity_mode	It sets a default mode for entity representation for all sessions opened from this SessionFactory
hibernate.order_updates	It orders SQL updates on the basis of the updated primary key.
hibernate.use_identifier_rollback	If enabled, the generated identifier properties will be reset to default values when objects are deleted.
hibernate.generate_statistics	If enabled, the Hibernate will collect statistics useful for performance tuning.
hibernate.use_sql_comments	If enabled, the Hibernate generate comments inside the SQL. It is used to make debugging easier.

Hibernate Cache Properties

Property	Description
hibernate.cache.provider_class	It represents the classname of a custom CacheProvider.
hibernate.cache.use_minimal_puts	It is used to optimize the second-level cache. It minimizes writes, at the cost of more frequent reads.
hibernate.cache.use_query_cache	It is used to enable the query cache.
hibernate.cache.use_second_level_cache	It is used to disable the second-level cache, which is enabled by default for classes which specify a mapping.
hibernate.cache.query_cache_factory	It represents the classname of a custom QueryCache interface.
hibernate.cache.region_prefix	It specifies the prefix which is used for second-level cache region names.

hibernate.cache.use_structured_entries	It facilitates Hibernate to store data in the second-level cache in a more human-friendly format.

Hibernate Transaction Properties

Property	Description
hibernate.transaction.factory_class	It represents the classname of a TransactionFactory which is used with Hibernate Transaction API.
hibernate.transaction.manager_lookup_class	It represents the classname of a TransactionManagerLookup. It is required when JVM-level caching is enabled.
hibernate.transaction.flush_before_completion	If it is enabled, the session will be automatically flushed during the before completion phase of the transaction.
hibernate.transaction.auto_close_session	If it is enabled, the session will be automatically closed during the after completion phase of the transaction.

Other Hibernate Properties

Property	Description
hibernate.connection.provider_class	It represents the classname of a custom ConnectionProvider which provides JDBC connections to Hibernate.
hibernate.connection.isolation	It is used to set the JDBC transaction isolation level.
hibernate.connection.autocommit	It enables auto-commit for JDBC pooled connections. However, it is not recommended.
hibernate.connection.release_mode	It specifies when Hibernate should release JDBC connections.
hibernate.current_session_context_class	It provides a custom strategy for the scoping of the "current" Session.
hibernate.hbm2ddl.auto	It automatically generates a schema in the database with the creation of SessionFactory.

JPA compliance

Property	Description
hibernate.jpa.compliance.transaction	This setting controls if Hibernate Transaction should behave as defined by the spec for JPA's javax.persistence.EntityTransaction since it extends the JPA one. (e.g. true or false (default value))
hibernate.jpa.compliance.query	Controls whether Hibernate's handling of javax.persistence.Query (JPQL, Criteria and native query) should strictly follow the JPA spec. This includes both in terms of parsing or translating a query as well as calls to the javax.persistence.Query methods throwing spec defined exceptions whereas Hibernate might not. (e.g. true or false (default value))
hibernate.jpa.compliance.list	Controls whether Hibernate should recognize what it considers a "bag" (org.hibernate.collection.internal.PersistentBag) as a List (org.hibernate.collection.internal.PersistentList) or as a bag. If enabled, we will recognize it as a List where javax.persistence.OrderColumn is just missing (and its defaults will apply). (e.g. true or false (default value))
hibernate.jpa.compliance.closed	JPA defines specific exceptions upon calling specific methods on javax.persistence.EntityManager and javax.persistence.EntityManagerFactory objects which have been closed previously.

	This setting controls whether the JPA spec-defined behavior or the Hibernate behavior will be used. If enabled, Hibernate will operate in the JPA specified way, throwing exceptions when the spec says it should. (e.g. true or false (default value))
hibernate.jpa.compliance.proxy	The JPA spec says that a javax.persistence.EntityNotFoundException should be thrown when accessing an entity proxy which does not have an associated table row in the database. Traditionally, Hibernate does not initialize an entity proxy when accessing its identifier since we already know the identifier value, hence we can save a database roundtrip. If enabled Hibernate will initialize the entity proxy even when accessing its identifier. (e.g. true or false (default value))
hibernate.jpa.compliance.global_id_generator	The JPA spec says that the scope of TableGenerator and SequenceGenerator names is global to the persistence unit (across all generator types). Traditionally, Hibernate has considered the names locally scoped. If enabled, the names used by @TableGenerator and @SequenceGenerator will be considered global so configuring two different generators with the same name will cause a java.lang.IllegalArgumentException to be thrown at boot time. (e.g. true or false (default value))