Hibernate Developer Guide

1

3.6.4.Final

by Gavin King, Christian Bauer, Max Rydahl Andersen,
Emmanuel Bernard, Steve Ebersole, and Hardy Ferentschik

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), Bernardo Antonio Buffa Colom&#x00e9, Vincent Ricard, Sebastien Cesbron, Michael Courcy, Vincent Giguère, Baptiste Mathus, Emmanuel Bernard, Anthony Patricio, Alvaro Netto, Anderson Braulio, Daniel Vieira Costa, Francisco gamarra, Gamarra, Luiz Carlos Rodrigues, Marcel Castelo, Paulo César, Pablo L. de Miranda, Renato Deggau, Rogério Araújo, Wanderson Siqueira, and Cao RedSaga Xiaogang

Preface v
1. Get Involved v
2. Getting Started Guide vi
1. Database Access 1
1.1. JDBC Connections 1
1.1.1. Using connection pooling 1
1.1.2. Using javax.sql.DataSource 2
1.2. Database Dialects 2
1.2.1. Specifying the Dialect to use 2
1.2.2. Dialect resolution 2
1.2.3. Custom Dialects 3
1.3. Database Schema 3

Preface

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping 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.

Database Access

1.1. JDBC Connections

Hibernate understands how to connect to a database through an interface `org.hibernate.connection.ConnectionProvider`. While `org.hibernate.connection.ConnectionProvider` is considered an extension SPI, Hibernate comes with a number of built-in providers.

1.1.1. Using connection pooling

The built-in connection pooling based providers all require the following settings

hibernate.connection.driver_class
: Names the `java.sql.Driver` implementation class from your JDBC provider.

hibernate.connection.url
: The JDBC connection url. See your JDBC provider's documentation for details and examples.

hibernate.connection.username
: The name of the user to use when opening a JDBC `java.sql.Connection`.

hibernate.connection.password
: The password associated with the provided username.

1.1.1.1. Using Hibernate's built-in connection pooling

Warning

The built-in Hibernate connection pool is not intended for production use. It lacks several features found on any decent connection pool. However, it can be quite useful to get started and in unit testing.

The only additional supported setting for the built-in pooling is the hibernate.connection.pool_size setting which tells the pool how many connections maximum it can keep in the pool.

1.1.1.2. Using c3p0 for connection pooling

To be continued...

1.1.1.3. Using proxool for connection pooling

To be continued...

1.1.2. Using `javax.sql.DataSource`

Hibernate can also use a `javax.sql.DataSource` to obtain connections. To do so, Hibernate expects to be able to locate the `javax.sql.DataSource` in `JNDI`. The hibernate.connection.datasource setting tells Hibernate the `JNDI` namespace at which it can find the the `javax.sql.DataSource`.

Generally speaking a `javax.sql.DataSource` is configured to connect to the database using a single set of credentials (username/password). Sometimes, however, the `javax.sql.DataSource` is set up so that the credentials have to be used to obtain a `java.sql.Connection` from it. In these cases applications would specify the credentials via the hibernate.connection.username and hibernate.connection.password settings, which Hibernate would pass along to the `javax.sql.DataSource` when obtaining a `java.sql.Connection` from it.

1.2. Database Dialects

A *Dialect* informs Hibernate of the capabilities of the underlying database. This role is fulfilled by an instance of a `org.hibernate.dialect.Dialect` subclass. The Dialect is one of the most important pieces of information given to the Hibernate `org.hibernate.Sessionfactory` during startup as it allows Hibernate to properly plan how it needs to communicate with the database.

1.2.1. 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 the specific `org.hibernate.dialect.Dialect` class to use.

1.2.2. Dialect resolution

Assuming a `org.hibernate.connection.ConnectionProvider` has been set up according to *Section 1.1, "JDBC Connections"* then 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.connection.ConnectionProvider`.

This functionality is provided by a series of `org.hibernate.dialect.resolver.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.dialect.resolver.DialectResolver` as follows

Example 1.1. Registering a custom `org.hibernate.dialect.resolver.DialectResolver`

```
org.hibernate.dialect.resolver.DialectFactory.registerDialectResolver( "org.hibernate.example.CustomDialectRes
```

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

1.2.3. Custom Dialects

It is sometimes necessary for developers to write a custom Dialect for Hibernate to use. Generally this is as simple as selecting a particular `org.hibernate.dialect.Dialect` implementation that is closest to your needs and subclassing it and overriding where necessary.

Custom dialects may be manually specified as outlined in *Section 1.2.1, "Specifying the Dialect to use"* as well as registered through a resolver as outlined in *Example 1.1, "Registering a custom org.hibernate.dialect.resolver.DialectResolver"*.

1.3. Database Schema

To be continued...