Hibernate.orgCommunity Documentation

Hibernate Getting Started Guide

The Hibernate Team

The JBoss Visual Design Team

4.1.6.Final

2012-08-09

Table of Contents

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 between an object model representation to a relational data model representation. See http://en.wikipedia.org/wiki/Object-relational_mapping for a good high-level discussion.

Note

You do not need a strong background in SQL to use Hibernate, but having a basic understanding of the concepts can help you understand Hibernate more fully and quickly. An understanding of data modeling principles is especially important. You might want to consider these resources as a good starting point:

Data Modeling Resources

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

Hibernate takes care of the mapping from Java classes to database tables, and from Java data types to SQL data types. In addition, it 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 streamlines the common task of translating result sets 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. Tutorial code

The referenced projects and code for the tutorials in this guide are available at files/hibernate-tutorials.zip.

Chapter 1. Obtaining Hibernate

Table of Contents

1.1. Release Bundle Downloads

The Hibernate team provides release bundles hosted on the SourceForge File Release System, in `ZIP` and `TGZ` formats. Each release bundle contains`JARs`, documentation, source code, and other goodness.

You can download releases of Hibernate, in your chosen format, from the list at http://sourceforge.net/projects/hibernate/files/hibernate4/.

- The `lib/required/` directory contains all the JARs Hibernate requires. All the jars in this directory must also be included in your project's classpath.
- The `/lib/jpa/` directory contains the `hibernate-entitymanager` jar and its dependencies beyond those in `lib/required/`. This defines Hibernate support for JPA.
- The `lib/envers` directory contains the `hibernate-envers` jar and its dependencies beyond those in `lib/required/`

- The `lib/optional` directory contains the jars needed for optional features of Hibernate.

1.2. Maven Repository Artifacts

Note

The authoritative repository for Hibernate artifacts is the JBoss Maven repository. The team responsible for the JBoss Maven repository maintains a number of Wiki pages that contain important information.

Maven Repository Wiki Pages

- http://community.jboss.org/docs/DOC-14900 - General information about the repository.
- http://community.jboss.org/docs/DOC-15170 - Information about setting up the JBoss repositories in order to do development work on JBoss projects themselves.
- http://community.jboss.org/docs/DOC-15169 - Information about setting up access to the repository to use JBoss projects as part of your own software.

Hibernate produces a number of artifacts (all under the `org.hibernate` groupId):

Hibernate Artifacts under groupId `org.hibernate`

hibernate-core

The main artifact, needed to build applications using the native Hibernate APIs including defining metadata in both annotations as well as Hibernate's own `hbm.xml` format.

hibernate-entitymanager

Represents Hibernate's implementation of JPA, as specified at http://jcp.org/en/jsr/detail?id=317.

This artifact depends on `hibernate-core`

hibernate-envers

An optional module that provides historical auditing of changes to your entities.

This artifact depends on both `hibernate-core` and `hibernate-entitymanager`.

hibernate-c3p0

Provides integration between Hibernate and the C3P0 connection pool library. See http://sourceforge.net/projects/c3p0/ for information about C3P0.

This artifact depends on `hibernate-core`, but is generally included in a project as a runtime dependency. It pulls in the C3P0 dependencies automatically.

hibernate-proxool

Provides integration between Hibernate and the Proxool connection pool library. See http://proxool.sourceforge.net/ for more information about this library.

This artifact depends on `hibernate-core`, but is generally included in a project as a runtime dependency. It pulls in the Proxool dependencies automatically..

hibernate-ehcache

Privides integration between Hibernate and EhCache, as a second-level cache. See http://ehcache.sourceforge.net/ for more information aboutEhCache.

This artifact depends on `hibernate-core`, but is generally included in a project as a runtime dependency. It pulls in the Ehcache dependencies automatically.

hibernate-infinispan

Provides integration between Hibernate and Infinispan, as a second-level cache. See http://jboss.org/infinispan for more information about Infinispan.

This artifact depends on `hibernate-core`, but is generally included in a project as a runtime dependency. It pulls in the Infinispan dependencies automatically.

Chapter 2. Tutorial Using Native Hibernate APIs and hbm.xml Mappings

Table of Contents

This tutorial is located within the download bundle under `basic/`.

Objectives

- using Hibernate mapping files (hbm.xml) to provide mapping information
- using the native Hibernate APIs

2.1. The Hibernate configuration file

The resource file `hibernate.cfg.xml` defines Hibernate configuration information.

The `connection.driver_class`, `connection.url`, `connection.username` and `connection.password` `property` elements define JDBC connection information. These tutorials utilize the H2 in-memory database, So the values of these properties are all specific to running H2 in its in-memory mode. `connection.pool_size` is used to configure the number of connections in Hibernate's built-in connection pool.

Important

The built-in Hibernate connection pool is in no way intended for production use. It lacks several features found on production-ready connection pools. See the section discussion in *Hibernate Developer Guide* for further information.

The `dialect` property specifies the particular SQL variant with which Hibernate will converse.

Tip

In most cases, Hibernate is able to properly determine which dialect to use. This is particularly useful if your application targets multiple databases. This is discussed in detail in the *Hibernate Developer Guide*

The `hbm2ddl.auto` property enables automatic generation of database schemas directly into the database.

Finally, add the mapping file(s) for persistent classes to the configuration. The `resource` attribute of the `mapping` element causes Hibernate to attempt to locate that mapping as a classpath resource, using a `java.lang.ClassLoader` lookup.

2.2. The entity Java class

The entity class for this tutorial is `org.hibernate.tutorial.hbm.Event`.

Notes About the Entity

- This class uses standard JavaBean naming conventions for property getter and setter methods, as well as private visibility for the fields. Although this is the recommended design, it is not required.
- The `no-argument` constructor, which is also a JavaBean convention, is a requirement for all persistent classes. Hibernate needs to create objects for you, using Java Reflection. The constructor can be private. However, package or public visibility is required for runtime proxy generation and efficient data retrieval without bytecode instrumentation.

2.3. The mapping file

The `hbm.xml` mapping file for this tutorial is the classpath resource `org/hibernate/tutorial/hbm/Event.hbm.xml` as we saw in Section 2.1, “The Hibernate configuration file”

Hibernate uses the mapping metadata to determine how to load and store objects of the persistent class. The Hibernate mapping file is one choice for providing Hibernate with this metadata.

Example 2.1. The `class` mapping element

```
<class name="Event" table="EVENTS">
    ...
</class>
```

Functions of the `class` mapping element

1. The `name` attribute (combined here with the `package` attribute from the containing `hibernate-mapping` element) names the FQN of the class to be defined as an entity.
2. The `table` attribute names the database table which contains the data for this entity.

Instances of the `Event` class are now mapped to rows in the EVENTS table.

Example 2.2. The `id` mapping element

```
<id name="id" column="EVENT_ID">
    ...
</id>
```

Hibernate uses the property named by the `id` element to uniquely identify rows in the table.

Important

It is not required for the `id` element to map to the table's actual primary key column(s), but it is the normal convention. Tables mapped in Hibernate do not even need to define primary keys. However, it is strongly recommend that all schemas define proper referential integrity. Therefore `id` and primary key are used interchangeably throughout Hibernate documentation.

The `id` element here identifies the EVENT_ID column as the primary key of the EVENTS table. It also identifies the `id` property of the `Event` class as the property containing the identifier value.

The `generator` element nested inside the `id` element informs Hibernate about which strategy is used to generated primary key values for this entity. This example uses a simple incrementing count.

Example 2.3. The `property` mapping element

```
<property name="date" type="timestamp" column="EVENT_DATE"/>
<property name="title"/>
```

The two `property` elements declare the remaining two properties of the `Event` class: `date` and `title`. The `date` property mapping includes the `column` attribute, but the `title` does not. In the absence of a `column` attribute, Hibernate uses the property name as the column name. This is appropriate for `title`, but since `date` is a reserved keyword in most databases, you need to specify a non-reserved word for the column name.

The `title` mapping also lacks a `type` attribute. The types declared and used in the mapping files are neither Java data types nor SQL database types. Instead, they are *Hibernate mapping types*. Hibernate mapping types are converters which translate between Java and SQL data types. Hibernate attempts to determine the correct conversion and mapping type autonomously if the `type` attribute is not present in the mapping, by using Java reflection to determine the Java type of the declared property and using a default mapping type for that Java type.

In some cases this automatic detection might not chose the default you expect or need, as seen with the `date` property. Hibernate cannot know if the property, which is of type `java.util.Date`, should map to a SQL DATE, TIME, or TIMESTAMP datatype. Full date and time information is preserved by mapping the property to a timestamp converter, which identifies an instance of the class `org.hibernate.type.TimestampType`.

Tip

Hibernate determines the mapping type using reflection when the mapping files are processed. This process adds overhead in terms of time and resources. If startup performance is important, consider explicitly defining the type to use.

2.4. Example code

The `org.hibernate.tutorial.hbm.NativeApiIllustrationTest` class illustrates using the Hibernate native API.

Note

The examples in these tutorials are presented as JUnit tests, for ease of use. One benefit of this approach is that `setUp` and `tearDown` roughly illustrate how a `org.hibernate.SessionFactory` is created at the start-up of an application and closed at the end of the application lifecycle.

Example 2.4. Obtaining the `org.hibernate.SessionFactory`

```
protected void setUp() throws Exception {
    // A SessionFactory is set up once for an application
    sessionFactory = new Configuration()
            .configure() // configures settings from hibernate.cfg.xml
            .buildSessionFactory();
}
```

Procedure 2.1. Tutorial Workflow

1. **The configuration is loaded.**

   The `org.hibernate.cfg.Configuration` class is the first thing to notice. In this tutorial, all configuration details are located in the `hibernate.cfg.xml` file discussed in Section 2.1, “The Hibernate configuration file”.

2. **The `org.hibernate.SessionFactory` is created.**

   The `org.hibernate.cfg.Configuration` then creates the `org.hibernate.SessionFactory` which is a thread-safe object that is instantiated once to serve the entire application.

3. **`SessionFactory` creates `Session` instances.**

   The `org.hibernate.SessionFactory` acts as a factory for `org.hibernate.Session` instances as can be seen in the `testBasicUsage` method.

4. **`Session`s perform work.**

   A `org.hibernate.Session` should be thought of as a corollary to a "unit of work".

Example 2.5. Saving entities

```
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save( new Event( "Our very first event!", new Date() ) );
session.save( new Event( "A follow up event", new Date() ) );
session.getTransaction().commit();
session.close();
```

`testBasicUsage` first creates some new `Event` objects and hands them over to Hibernate for management, using the `save` method. Hibernate now takes responsibility to perform an **INSERT** on the database.

Example 2.6. Obtaining a list of entities

```
session = sessionFactory.openSession();
session.beginTransaction();
List result = session.createQuery( "from Event" ).list();
for ( Event event : (List<Event>) result ) {
    System.out.println( "Event (" + event.getDate() + ") : " + event.getTitle() );
}
session.getTransaction().commit();
session.close();
```

`testBasicUsage` illustrates use of the *Hibernate Query Language (HQL)* to load all existing `Event` objects from the database and generate the appropriate `SELECT` SQL, send it to the database and populate `Event` objects with the result set data.

2.5. Take it further!

Practice Exercises

- Reconfigure the examples to connect to your own persistent relational database.
- With help of the *Developer Guide*, add an association to the `Event` entity to model a message thread.

Chapter 3. Tutorial Using Native Hibernate APIs and Annotation Mappings

Table of Contents

This tutorial is located within the download bundle under `basic`.

Objectives

- Use annotations to provide mapping information
- Use the native Hibernate APIs

3.1. The Hibernate configuration file

The contents are identical to Section 2.1, “The Hibernate configuration file”, with one important difference. The `mapping` element at the very end naming the annotated entity class using the `class` attribute.

3.2. The annotated entity Java class

The entity class in this tutorial is `org.hibernate.tutorial.annotations.Event` which follows JavaBean conventions. In fact the class itself is identical to the one in Section 2.2, "The entity Java class", except that annotations are used to provide the metadata, rather than a separate `hbm.xml` file.

Example 3.1. Identifying the class as an entity

```
@Entity
@Table( name = "EVENTS" )
public class Event {
    ...
}
```

The `@javax.persistence.Entity` annotation is used to mark a class as an entity. It functions the same as the `class` mapping element discussed in Section 2.3, "The mapping file". Additionally the `@javax.persistence.Table` annotation explicitly specifies the table name. Without this specification, the default table name would be `EVENT`).

Example 3.2. Identifying the identifier property

```
@Id
@GeneratedValue(generator="increment")
@GenericGenerator(name="increment", strategy = "increment")
public Long getId() {
    return id;
}
```

`@javax.persistence.Id` marks the property which defines the entity's identifier. `@javax.persistence.GeneratedValue` and `@org.hibernate.annotations.GenericGenerator` work in tandem to indicate that Hibernate should use Hibernate's `increment` generation strategy for this entity's identifier values.

Example 3.3. Identifying basic properties

```
public String getTitle() {
    return title;
}
```

```
@Temporal(TemporalType.TIMESTAMP)
@Column(name = "EVENT_DATE")
public Date getDate() {
    return date;
}
```

As in Section 2.3, “The mapping file”, the `date` property needs special handling to account for its special naming and its SQL type.

3.3. Example code

`org.hibernate.tutorial.annotations.AnnotationsIllustrationTest` is essentially the same as `org.hibernate.tutorial.hbm.NativeApiIllustrationTest` discussed in Section 2.4, “Example code”.

3.4. Take it further!

Practice Exercises

- Add an association to the `Event` entity to model a message thread. Use the *Developer Guide* as a guide.
- Add a callback to receive notifications when an `Event` is created, updated or deleted. Try the same with an event listener. Use the *Developer Guide* as a guide.

Chapter 4. Tutorial Using the *Java Persistence API (JPA)*

Table of Contents

This tutorial is located within the download bundle under `entitymanager`.

Objectives

- Use annotations to provide mapping information.
- Use `JPA`.

4.1. `persistence.xml`

The previous tutorials used the Hibernate-specific *`hibernate.cfg.xml`* configuration file. `JPA`, however, defines a different bootstrap process that uses its own configuration file named `persistence.xml`. This bootstrapping process is defined by the `JPA` specification. In Java™ SE environments the persistence provider (Hibernate in this case) is required to locate all `JPA` configuration files by classpath lookup of the `META-INF/persistence.xml` resource name.

Example 4.1. `persistence.xml`

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0">
    <persistence-unit name="org.hibernate.tutorial.jpa">
        ...
    </persistence-unit>
</persistence>
```

`persistence.xml` files should provide a unique name for each persistence unit. Applications use this name to reference the configuration when obtaining an `javax.persistence.EntityManagerFactory` reference.

The settings defined in the `properties` element are discussed in Section 2.1, “The Hibernate configuration file”. Here the `javax.persistence`-prefixed varieties are used when possible. Notice that the remaining Hibernate-specific configuration setting names are now prefixed with `hibernate.`.

Additionally, the `class` element functions the same as in Section 3.1, “The Hibernate configuration file”.

4.2. The annotated entity Java class

The entity is exactly the same as in Section 3.2, “The annotated entity Java class”

4.3. Example code

The previous tutorials used the Hibernate APIs. This tutorial uses the `JPA` APIs.

Example 4.2. Obtaining the `javax.persistence.EntityManagerFactory`

```
protected void setUp() throws Exception {
    entityManagerFactory = Persistence.createEntityManagerFactory( "org.hibernate.tutorial.jpa" );
}
```

Notice again that the persistence unit name is `org.hibernate.tutorial.jpa`, which matches Example 4.1, “`persistence.xml`”

Example 4.3. Saving (persisting) entities

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
entityManager.persist( new Event( "Our very first event!", new Date() ) );
entityManager.persist( new Event( "A follow up event", new Date() ) );
entityManager.getTransaction().commit();
entityManager.close();
```

The code is similar to Example 2.5, “Saving entities”. An `javax.persistence.EntityManager` interface is used instead of a `org.hibernate.Session` interface. `JPA` calls this operation `persist` instead of `save`.

Example 4.4. Obtaining a list of entities

```
entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
List<Event> result = entityManager.createQuery( "from Event", Event.class ).getResultList();
for ( Event event : result ) {
    System.out.println( "Event (" + event.getDate() + ") : " + event.getTitle() );
}
entityManager.getTransaction().commit();
entityManager.close();
```

Again, the code is pretty similar to Example 2.6, “Obtaining a list of entities”.

4.4. Take it further!

Practice Exercises

- Develop an EJB Session bean to investigate implications of using a container-managed persistence context (`javax.persistence.EntityManager`). Try both stateless and stateful use-cases.

Chapter 5. Tutorial Using Envers

Table of Contents

This tutorial is located within the download bundle under `envers`.

Objectives

- Configure Envers.
- Use the Envers APIs to view and analyze historical data.

5.1. `persistence.xml`

This file was discussed in the JPA tutorial in Section 4.1, "`persistence.xml`", and is essentially the same here.

5.2. The annotated entity Java class

Again, the entity is largely the same as in Section 4.2, "The annotated entity Java class". The major difference is the addition of the `@org.hibernate.envers.Audited` annotation, which tells Envers to automatically track changes to this entity.

5.3. Example code

Again, this tutorial makes use of the JPA APIs. However, the code also makes a change to one of the entities, then uses the Envers API to pull back the initial *revision* as well as the updated revision. A revision refers to a version of an entity.

Example 5.1. Using the `org.hibernate.envers.AuditReader`

```
public void testBasicUsage() {
    ...
    AuditReader reader = AuditReaderFactory.get( entityManager );
    Event firstRevision = reader.find( Event.class, 2L, 1 );
    ...
    Event secondRevision = reader.find( Event.class, 2L, 2 );
    ...
}
```

Procedure 5.1. Description of Example

1. An `org.hibernate.envers.AuditReader` is obtained from the `org.hibernate.envers.AuditReaderFactory` which wraps the `javax.persistence.EntityManager`.
2. Next, the `find` method retrieves specific revisions of the entity. The first call reads `find revision number 1 of Event with id 2`. The second call reads `find revision number 2 of Event with id 2`.

5.4. Take it further!

Practice Exercises

- Provide a custom revision entity to additionally capture who made the changes.
- Write a query to retrieve only historical data which meets some criteria. Use the *Envers User Guide* to see how Envers queries are constructed.
- Experiment with auditing entities which have many-to-one, many-to-many relations as well as collections. Try retrieving historical versions (revisions) of such entities and navigating the object tree.
