Hibernate User Guide

This page describes functionality provided by the Tapestry-hibernate-core module, but the descriptions apply equally to the Tapestry-jpa module.

Related Articles

- Hibernate - Core - Conf
- Hibernate - Core
- Hibernate Statistics
- Hibernate User Guide
- Hibernate
- Hibernate Support FAQ
- Using Tapestry With Hibernate

Entity value encoding

The Tapestry-hibernate-core module provides Value Encoder automatically for all mapped Hibernate entity types. This is done by encoding the entity as it's id (coerced to a String) and decoding the entity by looking it up in the Hibernate Session using the encoded id. Consider the following example:

Accessing the page as **/viewperson/152** would load the Person entity with id 152 and use that as the page context.

Using @PageActivationContext

If you prefer to use annotations, you may let Tapestry generate the page activation context handlers for you. Relying on an existing ValueEncoder for the corresponding property you can use the @PageActivationContext annotation. The disadvantage is that you can't access the handlers in a unit test.

Using @Persist with entities

If you wish to persist an entity in the session, you may use the "entity" persistence strategy:

This persistence strategy works with any Hibernate entity that is associated with a valid Hibernate Session by persisting only the id of the entity. Notice that no onPassivate() method is needed; when the page renders the entity is loaded by the id stored in the session.

Using @SessionState with entities

Added in 5.2

The default strategy for persisting Session State Objects is "session". Storing a Hibernate entity into a <HttpSession> is problematic because the stored entity is detached from the Hibernate session. Similar to @Persist("entity") you may use the "entity" persistence strategy to persist Hibernate entities as SSOs:

For this purpose you need to set the value of the symbol <HibernateSymbols.ENTITY_SESSION_STATE_PERSISTENCE_STRATEGY_ENABLED> to <true>:

Alternatively you can apply the "entity" persistence strategy to a single Hibernate entity:

Committing Changes

All Hibernate operations occur in a transaction, but that transaction is aborted at the end of each request; thus any changes you make will be *lost* unless the transaction is committed.

The correct way to commit the transaction is via the @CommitAfter annotation:

In this example, the Person object may be updated by a form; the form's success event handler method, onSuccess() has the @CommitAfter annotation.

Behind the scenes, the @CommitAfter annotation causes the HibernateSessionManager's commit() method to be executed before the method returns.

The transaction will be committed when the method completes normally.

The transaction will be *aborted* if the method throws a RuntimeException.

The transaction will be **committed** if the method throws a *checked* exception (one listed in the throws clause of the method).

Managing Transactions using DAOs

As your application grows, you will likely create a Data Access Object layer between your pages and the Hibernate APIs.

The @CommitAfter annotation can be useful there as well.

You may use @CommitAfter on method of your service interface, then use a decorator to provide the transaction management logic.

First definine your DAO's service interface:

Next, define your service in your application's Module class:

Finally, you should use the HibernateTransactionAdvisor to add transaction advice:

This advice method is configured to match against any service whose id ends with "DAO", such as "PersonDAO".

The advisor scans the service interface and identifies any methods with the @CommitAfter annotation.