Seam Persistence

1. Seam Persistence Reference 1
1.1. Introduction 1
1.2. Getting Started 1
1.3. Transaction Management 2
1.3.1. Configuration 2
1.3.2. Declarative Transaction Management 4
1.4. Seam-managed persistence contexts 5
1.4.1. Using a Seam-managed persistence context with JPA 5
1.4.2. Seam-managed persistence contexts and atomic conversations 6
1.4.3. Using EL in EJB-QL/HQL 6
1.4.4. Setting up the EntityManager 7

Seam Persistence Reference

Seam provides extensive support for the two most popular persistence architectures for Java: Hibernate3, and the Java Persistence API introduced with EJB 3.0. Seam's unique state-management architecture allows the most sophisticated ORM integration of any web application framework.

1.1. Introduction

Seam grew out of the frustration of the Hibernate team with the statelessness typical of the previous generation of Java application architectures. The state management architecture of Seam was originally designed to solve problems relating to persistence — in particular problems associated with *optimistic transaction processing*. Scalable online applications always use optimistic transactions. An atomic (database/JTA) level transaction should not span a user interaction unless the application is designed to support only a very small number of concurrent clients. But almost all interesting work involves first displaying data to a user, and then, slightly later, updating the same data. So Hibernate was designed to support the idea of a persistence context which spanned an optimistic transaction.

Unfortunately, the so-called "stateless" architectures that preceded Seam and EJB 3.0 had no construct for representing an optimistic transaction. So, instead, these architectures provided persistence contexts scoped to the atomic transaction. Of course, this resulted in many problems for users, and is the cause of the number one user complaint about Hibernate: the dreaded `LazyInitializationException`. What we need is a construct for representing an optimistic transaction in the application tier.

EJB 3.0 recognizes this problem, and introduces the idea of a stateful component (a stateful session bean) with an *extended persistence context* scoped to the lifetime of the component. This is a partial solution to the problem (and is a useful construct in and of itself) however there are two problems:

- The lifecycle of the stateful session bean must be managed manually via code in the web tier (it turns out that this is a subtle problem and much more difficult in practice than it sounds).
- Propagation of the persistence context between stateful components in the same optimistic transaction is possible, but tricky.

Seam solves the first problem by providing conversations, and stateful session bean components scoped to the conversation. (Most conversations actually represent optimistic transactions in the data layer.) This is sufficient for many simple applications (such as the Seam booking demo) where persistence context propagation is not needed. For more complex applications, with many loosly-interacting components in each conversation, propagation of the persistence context across components becomes an important issue. So Seam extends the persistence context management model of EJB 3.0, to provide conversation-scoped extended persistence contexts.

1.2. Getting Started

To get started with Seam persistence you need to add the `seam-persistence.jar` and the `seam-solder.jar` to you deployment. If you are in a java SE environment you will probably also require `seam-xml.jar` as well for configuration purposes. The relevant maven configuration is as follows:

```
<dependency>
  <groupId>org.jboss.seam.persistence</groupId>
  <artifactId>seam-persistence-api</artifactId>
  <version>${seam.persistence.version}</version>
</dependency>

<dependency>
  <groupId>org.jboss.seam.persistence</groupId>
```

```
  <artifactId>seam-persistence-impl</artifactId>
  <version>${seam.persistence.version}</version>
</dependency>

<dependency>
  <groupId>org.jboss.seam.solder</groupId>
  <artifactId>seam-solder</artifactId>
  <version>${seam.solder.version}</version>
</dependency>

<dependency>
  <groupId>org.jboss.seam.xml</groupId>
  <artifactId>seam-xml-config</artifactId>
  <version>${seam.xml.version}</version>
</dependency>
```

You will also need to have a JPA provider on the classpath. If you are using java EE this is taken care of for you. If not, we recommend hibernate.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.5.1-Final</version>
</dependency>
```

1.3. Transaction Management

Unlike EJB session beans CDI beans are not transactional by default. Seam brings declarative transaction management to CDI beans by enabling them to use `@TransactionAttribute`. Seam also provides the `@Transactional` annotation, for environments where java EE APIs are not present.

1.3.1. Configuration

In order to enable declarative transaction management for managed beans you need to list the transaction interceptor in beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://docs.jboss.org/cdi/beans_1_0.xsd">
   <interceptors>
      <class>org.jboss.seam.transaction.TransactionInterceptor</class>
   </interceptors>
</beans>
```

If you are in a Java EE 6 environment then you are good to go, no additional configuration is required.

If you are not in an EE environment you may need to configure some things with seam-xml. You may need the following entries in your beans.xml file:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:s="urn:java:ee"
   xmlns:t="urn:java:org.jboss.seam.transaction"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://docs.jboss.org/cdi/beans_1_0.xsd">

      <t:SeSynchronizations>
         <s:modifies/>
      </t:SeSynchronizations>

      <t:EntityTransaction>
         <s:modifies />
      </t:EntityTransaction>

</beans>
```

Let's look at these individually.

```
<t:SeSynchronizations>
  <s:modifies/>
</t:SeSynchronizations>
```

Seam will attempt to use JTA synchronizations if possible. If not then you need to install the `SeSynchronzations` bean to allow seam to handle synchronizations manually. Synchronizations allow seam to respond to transaction events such as `beforeCompletion()` and `afterCompletion()`, and are needed for the proper operation of the *Seam Managed Persistence Context*.

```
<t:EntityTransaction>
   <s:modifies />
</t:EntityTransaction>
```

By default seam will attempt to look up `java:comp/UserTransaction` from JNDI (or alternatively retrieve it from the `EJBContext` if a container managed transaction is active). Installing `EntityTransaction` tells seam to use the JPA `EntityTransaction` instead. To use this you must have a *Seam Managed Persistence Context* installed with qualifier `@Default`.

If your entity manager is installed with a different qualifier, then you need to use the following configuration (this assumes that `my` has been bound to the namespace that contains the appropriate qualifier, see the Seam Config XML documentation for more details):

```
<t:EntityTransaction>
   <s:modifies />
   <t:entityManager>
      <my:SomeQualifier/>
   </tentityManager>
</t:EntityTransaction>
```

Note

You should avoid `EntityTransaction` if you have more than one persistence unit in your application. Seam does not support installing multiple `EntityTransaction` beans, and the `EntityTransaction` interface does not support two phase commit, so unless you are careful you may have data consistency issues. If you need multiple persistence units in your application then we highly recommend using an EE 6 compatible server, such as JBoss 6.

1.3.2. Declarative Transaction Management

Seam adds declarative transaction support to managed beans. Seam re-uses the EJB `@TransactionAttribute` for this purpose, however it also provides an alternative `@Transactional` annotation for environments where the EJB API's are not available. An alternative to `@ApplicationException`, `@SeamApplicationException` is also provided. Unlike EJBs, managed beans are not transactional by default, you can change this by adding the `@TransactionAttribute` to the bean class.

Unlike in Seam 2, transactions will not roll back whenever a non-application exception propagates out of a bean, unless the bean has the transaction intercepter enabled.

If you are using seam managed transactions as part of the seam-faces module you do not need to worry about declarative transaction management. Seam will automatically start a transaction for you at the start of the faces request, and commit it before the render response phase.

Warning

`@SeamApplicationException` will not control transaction rollback when using EJB container managed transactions. If you are in an EE environment then you should always use the EJB API's, namely `@TransactionAttribute` and `@ApplicationException`.

Note

`TransactionAttributeType.REQUIRES_NEW` and `TransactionAttributeType.NOT_SUPPORTED` are not yet supported on managed beans. This will be added before seam-persistence goes final.

Let's have a look at some code. Annotations applied at a method level override annotations applied at the class level.

```
@TransactionAttribute /*Defaults to TransactionAttributeType.REQUIRED */
class TransactionaBean
{

   /* This is a transactional method, when this method is called a transaction
    * will be started if one does not already exist.
    * This behavior is inherited from the @TransactionAttribute annotation on
    * the class.
    */
    void doWork()
    {
      ...
```

```
    }

    /* A transaction will not be started for this method, however it      */
    /* will not complain if there is an existing transaction active.      */
    @TransactionAttributeType(TransactionAttributeType.SUPPORTED)
    void doMoreWork()
    {
      ...
    }

    /* This method will throw an exception if there is no transaction active when */
    /* it is invoked.                                                              */

    @TransactionAttributeType(TransactionAttributeType.MANDATORY)
    void doEvenMoreWork()
    {
      ...
    }

    /* This method will throw an exception if there is a transaction active when */
    /* it is invoked.                                                            */
    @TransactionAttributeType(TransactionAttributeType.NOT_SUPPORTED)
    void doOtherWork()
    {
      ...
    }
}
```

1.4. Seam-managed persistence contexts

If you're using Seam outside of a Java EE environment, you can't rely upon the container to manage the persistence context lifecycle for you. Even if you are in an EE environment, you might have a complex application with many loosely coupled components that collaborate together in the scope of a single conversation, and in this case you might find that propagation of the persistence context between component is tricky and error-prone.

In either case, you'll need to use a *managed persistence context* (for JPA) or a *managed session* (for Hibernate) in your components. A Seam-managed persistence context is just a built-in Seam component that manages an instance of `EntityManager` or `Session` in the conversation (or any other) context. You can inject it with `@Inject`.

1.4.1. Using a Seam-managed persistence context with JPA

```
@SeamManaged
@Produces
@PersistenceUnit
@ConversationScoped
EntityManagerFactory producerField;
```

This is just an ordinary resource producer field as defined by the CDI specification, however the presence of the `@SeamManaged` annotation tells seam to create a seam managed persistence context from this `EntityManagerFactory`. This managed persistence context can be injected normally, and has the same scope and qualifiers that are specified on the resource producer field.

This will work even in a SE environment where `@PersistenceUnit` injection is not normally supported. This is because the seam persistence extensions will bootstrap the `EntityManagerFactory` for you.

Now we can have our `EntityManager` injected using:

```
@Inject EntityManager entityManager;
```

Note

The more eagle eyed among you may have noticed that the resource producer field appears to be conversation scoped, which the CDI specification does not require containers to support. This is actually not the case, as the @ConversationScoped annotation is removed by the seam persistence portable extension. It only specifies the scope of the created SMPC, not the EntityManagerFactory.

Warning

If you are using EJB3 and mark your class or method `@TransactionAttribute(REQUIRES_NEW)` then the transaction and persistence context shouldn't be propagated to method calls on this object. However as the Seam-managed persistence context is propagated to any component within the conversation, it will be propagated to methods marked `REQUIRES_NEW`. Therefore, if you mark a method `REQUIRES_NEW` then you should access the entity manager using @PersistenceContext.

1.4.2. Seam-managed persistence contexts and atomic conversations

Persistence contexts scoped to the conversation allows you to program optimistic transactions that span multiple requests to the server without the need to use the `merge()` operation , without the need to re-load data at the beginning of each request, and without the need to wrestle with the `LazyInitializationException` or `NonUniqueObjectException`.

As with any optimistic transaction management, transaction isolation and consistency can be achieved via use of optimistic locking. Fortunately, both Hibernate and EJB 3.1 make it very easy to use optimistic locking, by providing the `@Version` annotation.

By default, the persistence context is flushed (synchronized with the database) at the end of each transaction. This is sometimes the desired behavior. But very often, we would prefer that all changes are held in memory and only written to the database when the conversation ends successfully. This allows for truly atomic conversations. Unfortunately there is currently no simple, usable and portable way to implement atomic conversations using EJB 3.1 persistence. However, Hibernate provides this feature as a vendor extension to the `FlushModeTypes` defined by the specification, and it is our expectation that other vendors will soon provide a similar extension.

1.4.3. Using EL in EJB-QL/HQL

Seam proxies the `EntityManager` or `Session` object whenever you use a Seam-managed persistence context. This lets you use EL expressions in your query strings, safely and efficiently. For example, this:

```
User user = em.createQuery("from User where username=#{user.username}")
   .getSingleResult();
```

is equivalent to:

```
User user = em.createQuery("from User where username=:username")
   .setParameter("username", user.getUsername())
   .getSingleResult();
```

Of course, you should never, ever write it like this:

```
User user = em.createQuery("from User where username=" + user.getUsername()) //BAD!
   .getSingleResult();
```

(It is inefficient and vulnerable to SQL injection attacks.)

Warning

This only works with seam managed persistence contexts, not persistence contexts that are injected with `@PersistenceContext`.

1.4.4. Setting up the EntityManager

Sometimes you may want to perform some additional setup on the `EntityManager` after it has been created. For example, if you are using Hibernate you may want to set a filter. Seam persistence fires a `SeamManagedPersistenceContextCreated` event when a Seam managed persistence context is created. You can observe this event and perform any setup you require in an observer method. For example:

```
public void setupEntityManager(@Observes SeamManagedPersistenceContextCreated event) {
   Session session = (Session)event.getEntityManager().getDelegate();
   session.enableFilter("myfilter");
}
```