Rice 2.x - JPA Approach

Overview

This document describes some of the main issues we've encountered in migration Kuali Rice and KNS to JPA as well as some possible approaches to solving these. This document is not meant to be a final design document, but rather a way to facilitate discussion and further thought about how to address these different issues.

Terminology

To help set the stage for the rest of this document, we will first define some JPA terminology along with some examples:

- **Entity** - a POJO who's non-transient fields should be persisted to a relational database by a JPA provider
- **Persistence Unit** - a named configuration of entities
- **Persistence Context** - a managed set of entity instances, every Persistence Context is associated with a Persistence Unit
- **EntityManager** - provides an API to manage a Persistence Context
- **Detached Entity** - an entity who is no longer associated with a Persistence Context

EntityManagers

There are two main classifications of Entity Managers:

- **Container-Managed**- the lifecycle of the EntityManager is handled by the "container", typically in a J2EE environment. There are two types of Container-Managed EntityManagers:
  - Transaction-Scoped - the persistence context is created automatically or retrieved from current active JTA transaction
    - ```
      @PersistenceContext(unitName="puName")
      EntityManager entityManager;
      ```
  - Extended - designed specifically for StatefulSessionBeans in EJB and used with a StatefulSessionBean "conversation"
    - entities are not detached when the transaction ends
    - application code needs to keep a reference around to the EntityManager in order to use it across multiple calls on the StatefulSessionBean
      - ```
        @PersistenceContext(unitName="puName",
        type=PersistenceContextType.EXTENDED)
        EntityManager entityManager;
        ```
- **Application-Managed**- the application manually manages the EntityManager using an EntityManagerFactory
  - ```
    EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("puName");
    EntityManager entityManager = emf.createEntityManager();
    ```
    ```

- The EntityManager needs to be explicitly closed by the application
- This makes it similar in nature to an Extended Container-Managed persistence context

Transaction Management

JPA supports two different types of transactions:

1. resource-local - uses the native transaction of the JDBC drivers referenced by a persistence unit (i.e. by calling commit, rollback, etc. on the underlying java.sql.Connection)
2. JTA - transactions provided by the J2EE container, allows for transactions to span multiple databases (and therefore, multiple Persistence Units)

Container-managed entity managers always use JTA transactions. Application-managed entity managers can use both.

In terms of JTA transactions with JPA, there is some additional terminology to understand which is useful:

- **Transaction Synchronization** - the process by which a persistence context is registered with a transaction so that the persistence context can be notified with the transaction commits.
- **Transaction Association** - the act of binding a persistence context to a transaction
- **Transaction Propogation** - the process of sharing a persistence context between multiple container-managed entity managers in a single transaction

There is no limit to the number of application-managed persistence contexts that can be synced with a transaction, but only one container-managed persistence context for a given persistence unit will ever be associated. This is one of the most important differences between application-managed and container-managed persistence contexts.

Application-managed persistence contexts can participate in JTA transactions as well in one of two ways:

1. If persistence context created inside of a transaction, the provider will auto-sync the persistence context with the transaction
2. If the persistence context was created earlier, it can join to the current JTA transaction using EntityManager.joinTransaction().

JPA Challenge Areas

The main challenge areas with JPA that this document will discuss include:

1. Entity Detachment, Merging, and Lazy Loading
2. Identifier Generation
3. Type Conversion
4. Criteria-based Queries
5. Metadata/Metamodel
6. Extension Framework

Entity Detachment, Merging and Lazy Loading

One of the biggest problems we've encountered is that of Entity detachment. With a transaction-scoped persistence context, all entities are detached when the transaction commits or rolls back. With an extended or application-managed persistence context, entities are detached when the entity manager is closed **or** whenever the current transaction is rolled back (anytime a rollback occurs, all entities are detached).

Additionally, once an entity is detached, lazy loading of relationships or collection elements which have not yet been materialized will have an **unspecified behavior** from the perspective of the JPA spec (the spec makes no mention of what should happen in this case). For example, in the case of Hibernate a `LazyInitializationException` is thrown but this is a Hibernate-specific exception.

We have to deal with this problem in KRAD because frameworks like the document framework will pull entities into the HTTP session and hang onto them across multiple requests, applying changes initiated by the user to the entity until the final submission of the form (at which case a workflow process is invoked, ultimately wanting to save the entity to the database). The issue here is that if the entity is detached then lazy loading of additional attributes will not work as desired. We can merge the entity back into a new persistence context prior to saving, but merges don't cascade automatically unless specified in the annotation on the relationship.

Obviously, this kind of requirement is common within the web application world. With JPA, there are a few common approaches for dealing with this overall problem:

1. Session Facade - keep detached entity in the HTTP session, apply changes to that, then merge them to a new persistence context when ready to persist.
2. Edit Session - use an extended persistence context and bind the EntityManager to the HTTP Session. That way the entities remain attached across multiple requests. Needs to be cleaned up at the end (necessitating the need for well-defined boundaries), otherwise will stick around until the session expires.
3. Transfer Objects - translate entities to some other form (such as transfer objects) for editing up at the web layer, and then translate back down during save back to database.

When using an approach like Session Facade, planning for the impending detachment needs to be done. The most important aspect of this is

triggering lazy loading for attributes which may need to be utilized in the detached entity. Otherwise, when attempting to access those attributes of the entity once it is detached attempting to access a non-materialized lazy attribute will result in undefined behavior.

Recommendations:

The *Session Facade* approach is cited as the most common approach to dealing with this problem. KRAD should really be able to support this approach as well as a service-level facade using transfer objects (which it already supports to some extent as evidenced by Kuali Student). An application which is using the UIF in KRAD to build it's user interface will define information within the Data Dictionary that can be used to make "eager-fetching" decisions. Therefore, as part of this effort we should implement the ability within KRAD to trigger lazy loading of any attributes which are lazy and might be needed in the view. This should be done prior to detachment. It would also be worthwhile to have an available hook in the framework that allows for manual triggering of lazy attribute loading prior to detachment (for cases where the framework cannot automatically detect it).

POC Work:

- Create a simple POC which loads an entity from the database, stores it in session, modifies it over multiple request response cycles, and then merges it back down at the end. We should also include a sufficiently complex entity that includes lazy-loaded attributes as well and try to test as many different scenarios as possible. Use Spring MVC as the framework for testing this scenario.
- Verify that the JPA metamodel includes information that can be used to determine which relationships have a LAZY fetch type.
- Branch Rice 2.2 and create a POC inside of Rice which attempts to implement the session facade approach within the KRAD maintenance framework.

Update: EclipseLink will handle this issue for us with no additional work. Given this will be our provider out of the box no additional work will be needed in this area. However, we are considering providing a 'conversation' hook with a entity manager wrapper so use with hibernate will function correctly within the framework.

Identifier Generation

JPA provides 4 built-in strategies for automatic identifier generation. The different types are as follows:

- **AUTO** - the provider will use whatever strategy it wants to generate identifiers, meant primarily for development or prototyping
- **TABLE** - uses a table for tracking and generating identifiers, locking semantics when using this approach are provider-specific and undefined in the JPA spec
- **SEQUENCE** - generates identifiers using a named database sequence, only works on databases that support sequences
- **IDENTITY** - uses an identity column in the database table (such as a column with the AUTO_INCREMENT attribute in MySQL)

These are enabled using the `@Generated` annotation in conjunction with the enumerated strategy to use, as in the following example:

```
@Entity
public class MyClass {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    // ...

}
```

It does **not** allow for the implementation of custom strategies. Unfortunately for Rice (and other Kuali applications based on the KNS), we use a custom approach for how we handle MySQL "sequences" using OJB so we are going to need to determine which approach will work best.

Recommendations:

With the exception of the **TABLE** strategy, there is no way in JPA way to handle identifier generation which is portable across different databases (i.e. Oracle, MySQL, etc.). The table-based approach is similar to what we do with MySQL but suffers from the issue of locking semantics being provider-specific and undefined as part of the JPA specification. In part, this is because the table-based approach uses a single row to manage to current identifier value. A naive implementation would issue an update that increments the value by 1. This would result in that row getting locked and would essentially result in a significant number of deadlock scenarios. I'm sure that actual JPA vendor implementations of the TABLE strategy do not have such an implementation but rather grab id values in "chunks". Depending on the chunk size and system load, however, this could still result in deadlock unless new id ranges were grabbed outside of the current transaction. With our MySQL approach in Kuali, we were careful to use a table-per-sequence approach which used an `auto_increment` column because the establishment of a new auto_increment value does not lock the table at all. Also, inserting a new row each time ensured that we weren't updating a shared row and therefore prevented row-level locking.

Regardless, in Kuali applications we have the current legacy implementation to deal with and so we must find an approach with supports that. There are two main ways that we can approach this:

1. Use the `@PrePersist` annotation or an `EntityListener` to pull a value from a Sequence (or our simulated sequences in MySQL) prior to saving the value
2. Require the `persist` operation to go through the `BusinessObjectService` and provision and apply the generated ID at that point

The first method has the advantage of being a (mostly) pure-JPA approach and would allow for the entity to be saved using the standard EntityManager API while still acquiring a PK in the process. There's one important caveat to this approach however, the JPA 2 specification states the following:

> *In general, the lifecycle method of a portable application should not invoke EntityManager or Query operations, access other entity instances, or modify relationships within the same persistence context.*

In the case of certain JPA vendors (Hibernate specifically) this behavior is strictly disallowed. So this means in order to pull a new identifier from the sequence, it would need to be done outside of the currently executing EntityManager (for example, through use of our non-transactional datasource).

The second method has the disadvantage that if you want to persist the entity, you have to either use the BusinessObjectService, or manually acquire the next value from the sequence. However, it does appear to provide a more portable solution to the problem.

Furthermore, we do have an existing solution that was put into place during Rice 1.1 development which uses `OrmUtils.populateAutoIncValue(...)`. This needs to be looked at to determine whether or not this original solution is still viable/desirable.

Type Conversion

JPA 2.0 has no way to provide type conversion (for example, if you store a "Y" or "N" in the database but want to use it as a boolean in the application). The only portable way to deal with this is to manually translate the value inside of the entity class.

However, it does look like JPA 2.1 will have support for converters. So manual conversion can be done as a workaround today with JPA 2.0 until JPA 2.1 is completed:

- http://wiki.eclipse.org/EclipseLink/Development/JPA_2.1

Update: EclipseLink provides type convertors that we can provide to applications for use. In order to not couple our code to EclipseLink we can provide these through an optional mapping file.

Criteria-based Queries

As of JPA 2.0, JPA supports a criteria API for queries which is very feature-rich. It should be possible to replace our hand-rolled JPA criteria api that is currently in Rice (from long ago) in favor of the native JPA 2.0 criteria API.

Update: The JPA 2.0 Criteria object will support our needs within the framework.

Metadata/Metamodel

In JPA 2.0, access to the metadata is available through the JPA "metamodel".

Update: The Metamodel provides a lot of information but not everything we need. We will need to get the rest by dropping down to the provider APIs.

Extension Framework

The issue with the extension framework is that in OJB-based KNS/KRAD, the extension object is on the PersistableBusinessObjectBase but is not mapped in OJB by default. It is not possible to do this in JPA using annotation. One original solution we looked at was dropping to XML mapping, which was somewhat unfortunate. However, it looks like perhaps that if the extension object is in the same persistence unit as the parent object, it will be possible to do joins with JPQL or the criteria API even if the extension object is not mapped directly from it's parent object. We can then manually fetch the extension object using the JPA api.

Update: Using vendor-specific apis we can modify the JPA mapping programatically, this will allow us to mix in extension mappings later programatically. JPA 2.1 purports to add support for programattic metadata mapping but it remains to be seen if that will make it into the final version or not.

JPA Provider Selection

Recommendation: Internal Rice modules should **not** use any vendor-specific extensions. If that is required anywhere, then we need to use an alternative means (such as straight JDBC). The reason for this is because some of the Rice modules can be embedded into client applications, at which point they will likely use the JPA implementation being used in the client application. So keeping Rice portable to JPA implementation will be critical.

Update: It turns out that it's possible to run different persistence units with different JPA providers. That said, the consensus from research done is that it will be best to use EclipseLink as the recommended JPA implementation since it's lazy loading semantics are very similar to OJB.

is it possible to run different persistence units with different persistence providers in the same jvm?

KRAD Data Access Layer Design

Design details can be found here: KRAD Data Layer Design

Related Roadmap Items

-
- KRRM-1 - Data cannot be retrieved due to an unexpected error