Flow-managed persistence in Spring Web Flow 2

Persistence strategies for transactional Web flows

Skill Level: Intermediate

Xinyu Liu (dr.xyliu@gmail.com)
Senior App Dev Consultant
Magellan Health Services

13 Apr 2010

Spring Web Flow 2's JPA/Hibernate persistence architecture is founded on the concept of flow-managed persistence, which before now has been only briefly documented. In this in-depth article, Xinyu Liu walks you through the conceptual building blocks of flow-managed persistence and the flow-scoped persistence context. He then demonstrates transactional strategies for handling atomic and non-atomic Web flows in complex, real-world development scenarios.

Spring Web Flow is an innovative Java™ Web framework that extends Spring MVC technology. Application development with Spring Web Flow is organized around use cases that are defined as Web flows. Having the development workspace organized in terms of Web flows results in a more meaningful and contextual development experience. In addition, Spring Web Flow's support for JPA/Hibernate persistence is one of its most substantial server-side contributions.

Although Spring Web Flow is well documented by SpringSource and the Spring Web Flow project team, its persistence support, and particularly its flow-managed persistence mechanism, are little understood. This article is an in-depth discussion of Java persistence programming in Spring Web Flow 2, focusing on flow-managed persistence and its essential component — the flow-scoped persistence context.

Following an overview of Spring Web Flow persistence concepts, I present use cases that demonstrate strategies for handling read-only and read/write transactions in atomic and non-atomic Web flows. In each case, I explain the conceptual underpinning of the preferred transaction-handling strategy and also demonstrate its shortcomings. The article concludes with my own guidelines for managing

transactions effectively and safely in Spring Web Flow 2.

This article is intended for experienced Java developers familiar with Spring Web Flow 2 and its continuations-based architecture. Use cases and sample application code will especially benefit developers already using JPA/Hibernate in their Spring Web Flow applications.

Persistence challenges in JPA/Hibernate

In a typical Web application, a user request is processed in two major steps: action handling and view rendering. The application's primary business logic resides in action handling. View rendering, which happens afterward, feeds data into a view template to generate a presentation.

In JPA/Hibernate, data (more specifically, entity relationships) may be eagerly loaded or lazy-loaded as proxy objects. If the persistence-context object (a JPA `EntityManager` or Hibernate `Session`) has already been closed in the view-rendering phase, entities become detached. Any attempt to access the unloaded relationships on the detached entities will lead to a `LazyInitializationException`.

The *Open Session in View* pattern (see Resources) seeks to solve the problem of the `LazyInitializationException`. When Open Session in View is implemented as a filter or interceptor, the persistence-context object remains open during view rendering. Navigating to an unloaded relationship on a persistent entity will trigger additional database queries to fetch the relationship on demand.

A downside of the Open Session in View pattern is that the persistence-context object is effectively scoped to a user request. Consequently, entities stored in Servlet scopes other than the current request are always detached. Detached entities require merge/reattach/reload operations to be associated with the current persistence context.

Spring Web Flow takes a different approach to saving the detached-entity state trouble via flow-managed persistence and, more specifically, the flow-scoped persistence-context object.

Flow-managed persistence

Application development in Spring Web Flow is based on the concept of the *Web flow*, which typically represents a sole use case. In many instances, data changes throughout a Web flow are required to be *atomic*, meaning that changes made at different stages of the flow either save as a whole to the backend database, or cancel out completely with no trace left in the database.

Spring Web Flow facilitates JPA/Hibernate programming in transactional atomic Web flows via the mechanism of *flow-managed persistence*. Flow-managed persistence is conceptually identical to Hibernate/Seam conversations (see Resources), where data changes made during a Web flow (or "page flow" in Seam) are cached in the same flow-scoped persistence-context object as dirty entities. No SQL insert/update/delete statements are fired until the end of the flow, when changes are flushed and committed to the database all at once. (Note that "flush" and "commit" are distinct concepts: the former fires a series of SQL insert/update/delete statements to synchronize dirty entities with their corresponding database values, whereas the latter just commits a database transaction.)

OptimisticLockingFailureException in flow-managed persistence

Optimistic locking is an extremely efficient concurrency control method that guarantees data integrity without placing any physical lock in the database. While not enforced, optimistic locking is highly recommended in flow-managed persistence.

The persistence context checks entity versions at flush time, throwing an `OptimisticLockingFailureException` (`StaleObjectException` in Hibernate) if it detects concurrent modification. The longer an entity lives in memory, the more likely its corresponding database value will be modified by other processes.

In the Open Session in View pattern, as previously mentioned, the persistent state of the entities is subject to the user request. Once an entity becomes detached, a merge/reattach/reload operation is generally required in subsequent user requests to restore the persistent state of that entity, as a result, the entity and its corresponding database value are synchronized.

Entities keep their persistent state across multiple user requests in flow-managed persistence. Database synchronization is not mandated between user requests, therefore, there is a higher chance of running into the `OptimisticLockingFailureException`. The trick is to handle the `OptimisticLockingFailureException` gracefully, just as you would any checked business exception. (This is true even though `OptimisticLockingFailureException` is a runtime exception that rolls back the database transaction.) Common strategies are to present the user with an opportunity to merge his changes or to restart the flow with non-stale data.

The flow-scoped persistence context

A Web flow is declared as an XML-formatted flow-definition file. When a Web flow marked with a `<persistence-context/>` tag starts, a new persistence-context object is created and bound to the flow scope. The object is disconnected from the

underlying JDBC connection when waiting for a user request and reconnected when serving a user request. The same persistence-context object is reused during the course of the entire flow, eliminating detached-entity state and the corresponding `LazyInitializationException`.

The persistence context is also bound to the current request thread and exposed to developers in two different flavors: as an implicit variable `persistenceContext` or injected into any Spring bean via the JPA `@PersistenceContext` annotation.

The implicit variable is available directly in flow-definition XML files — for instance:

```
<evaluate expression="persistenceContext.persist(transientEntityInstance)"/>
```

The injected JPA entity manager may be referenced anywhere in Spring components, such as in DAOs, service beans, or Web-tier beans.

Types of persistence context: Transaction or extended

The `@PersistenceContext` annotation has an optional attribute `type`, which defaults to `PersistenceContextType.TRANSACTION` (that is, a transaction-bound persistence context). You have to use this default setting when programming with a flow-scoped persistence context. In that case the injected transaction-bound persistence-context object is just a shared proxy that transparently delegates to the actual thread-bound persistence context of the flow scope.

Selecting the other option, `PersistenceContextType.EXTENDED`, results in a so-called "extended entity manager", which is not thread-safe and must not be used in concurrently accessed components such as singleton Spring beans. Using an extended entity manager as your flow-scoped persistence context will lead to unpredictable database/transaction behavior in your applications, so avoid it.

Interestingly, a Seam conversation is typically implemented with an extended entity manager injected into a stateful session bean (EJB). That is a noticeable difference between Spring Web Flow's flow-managed persistence and a Seam conversation.

The flow-scoped persistence-context object may be used in conjunction with the `@Transactional` annotation to fine-tune the persistence characteristics of a flow.

Transaction semantics

The `@Transactional` annotation, part of the Spring Core package, specifies the transaction semantics of the annotated class or method. According to the Spring development team, `@Transactional` is better applied to concrete classes than

interfaces. Default transaction semantics are:

```
@Transactional(readOnly=false,propagation=PROPAGATION_REQUIRED,
  isolation=ISOLATION_DEFAULT,timeout=TIMEOUT_DEFAULT)
```

readOnly: Setting up a read/write transaction by specifying `@Transactional(readOnly=false)` will make the `FlushMode` of the persistence context `AUTO`. Applying `@Transactional(readOnly=true)` will make the `FlushMode` of the underlying Hibernate session `MANUAL`.

JPA 1.0 does not support `MANUAL` flush nor read-only transactions, so `@Transactional(readOnly=true)` is only meaningful when the underlying JPA provider, like Hibernate, supports read-only database transactions. Moreover, Hibernate uses this setting as a database hint against certain database types for optimized query performance.

propagation: The `propagation` attribute determines whether the current method is running under an inherited transaction, or in a new transaction by suspending/resuming the enclosing transaction, or in no transaction at all.

isolation: JPA 1.0 doesn't support custom isolation levels, so developers are required to specify the default transaction isolation level on the database side. `Read-Committed` is the minimum level required for optimistic locking to work.

timeout: The `timeout` attribute specifies how long the transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).

rollbackFor, rollbackForClassname, noRollbackFor, noRollbackForClassname: As a general rule, a transaction always rolls back on a `RuntimeException` representing a system error and commits on a checked `Exception` with a predefined business meaning. It is possible to customize the default semantics through these four rollback attributes.

The Spring Core's robust transaction infrastructure makes transaction management easier in most real-world development scenarios. In the sections that follow, we'll see how Spring Web Flow makes use of the Spring transaction infrastructure in conjunction with its own flow-scoped persistence-context object to handle persistence programming in a variety of Web flows, including some use cases that demonstrate the limitations of flow-managed persistence.

Atomic Web flows

Flow-managed persistence is intended to address Spring Web Flow use cases that are considered atomic from the perspective of transactions. For instance, let's say

that you have an online banking system that allows a user to move his money from a checking account to a savings account or a to-be-created CD account. The transaction has to be conducted in several steps:

1. The user selects a checking account to be transferred.
2. The system displays the account balance.
3. The user enters the amount to be transferred.
4. The user selects a savings or CD account as the target.
5. The system displays a summary of the transaction for review.
6. The user decides to commit or cancel the transaction.

Due to the obvious concurrency requirement, you would first enable optimistic locking on entity classes. For this you could use the JPA `@Version` annotation or a Hibernate proprietary `OptimisticLockType.ALL` attribute. You could then map the complete use case into a single Web flow marked with Spring Web Flow's `<persistence-context/>` tag.

Non-transactional data access in Web flows

In Spring Web Flow, all data access, by default, is non-transactional. For non-transactional data access, Hibernate sets the `auto-commit` mode of the underlying database to `true`, such that each SQL statement is immediately executed in its own "short transaction", commit or rollback. From an application perspective, a database short transaction is equivalent to no transaction at all. More critical, Hibernate disables the default `FlushMode.AUTO` for non-transactional operations. It effectively works as `FlushMode.MANUAL`.

Disabling `FlushMode.AUTO` is crucial to flow-managed persistence. Entity lazy-reads in the view-rendering phase are also executed in non-transactional mode. If a flush ever happened during the rendering of different views, there would be no chance to accomplish the deferred flush at the end of the flow. In essence, non-transactional reads in `auto-commit` mode are equivalent to reads within a transaction of isolation-level `Read-Committed`. Similarly, flush never happens to non-transactional write operations.

In the above use case, each user action can be executed outside of a database transaction, without the `@Transactional` annotation or XML-configured transaction advisors specified. The flow-scoped persistence-context object manages data loaded during the flow as persistent entities, and data changes cached as the entities' dirty states.

If a user confirms the money-transfer transaction at the end of the flow, via `<end-state commit="true"/>`, the Spring Web Flow runtime will call `entityManager.flush()` implicitly within a read/write database transaction. It will then commit the transaction, unbind the persistence context, and close it. If the user chooses to cancel the transaction via `<end-state commit="false"/>`, all cached data changes will be discarded from memory upon the closing of the flow-scoped persistence context.

This approach to flow-managed persistence matches exactly how JPA 1.0 interprets the handling of a conversation. The `JpaFlowExecutionListener` class is the underlying Spring Web Flow component that makes it happen. Apart from the non-transactional data access approach to flow-managed persistence, the alternative is to use read-only transactions.

Read-only transactions in Web flows

In some cases, you might prefer read-only transactions over non-transactional ones. If you look at the sample "Hotel Booking" application in Spring Web Flow releases (see Resources), you'll notice that `@Transactional(readOnly=true)` is used universally for all data access during the "booking" Web flow, regardless of the read/insert/update/delete nature of the operations.

Read-only transactions are not supported by the JPA 1.0 specification, so this setting is useable only in some JPA providers. In its JPA implementation, Hibernate sets the `FlushMode` of the underlying Hibernate session to `MANUAL` and the `auto-commit` mode to `false`.

Effectively, read-only transactions to flow-managed persistence act just like non-transactional data access, in that changed entities are only flushed at the end of an atomic Web flow via `<end-state commit="true"/>`.

If you want a flush to happen prior to the `<end-state/>`, you'll need to invoke `entityManager.flush()` in one of your Spring bean's methods, annotated with `@Transactional`.

A direct call from the Web flow, `<evaluate expression="persistenceContext.flush()"/>`, wouldn't work, as no transaction is bound to any Spring Web Flow tag other than `<end-state commit="true"/>`. You would get the following error message:

```
"javax.persistence.TransactionRequiredException: no transaction is in progress"
```

We'll return to the "Hotel Booking" example later in the article, for a look at the challenges of persistence programming without the flow-scoped persistence context.

More about transaction propagation

I've touched on how a transaction is propagated based on the value of its `propagation` attribute, but I ignored one particular use case: If a method marked `@Transactional(readOnly=true, propagation=Propagation.REQUIRED)` were to call another method marked `@Transactional(readOnly=false, propagation=Propagation.REQUIRED)`, or the other way around, then how would the transaction propagate?

Spring Web Flow handles this in a simple but elegant way: it ignores the `readOnly` attribute value on the second method. To put it simply, a transaction initiated as read-only remains read-only until it ends, and *vice versa*.

This has interesting implications for the question of whether to use no transactions or read-only transactions in flow-managed persistence.

A use case for read-only transactions

Spring beans in an application's service layer can be exposed as reusable SOAP/REST Web services through some JAX-WS/JAX-RS annotations. Applying `@Transactional` on these `@Service` beans or their methods binds the Web services invocations with database transactions. (There are no obvious reasons to use `@Transactional` on DAO `@Repository` beans, unless the application has a collapse layer architecture, where there are no other places for developers to specify transaction properties.)

Think again about the non-transactional data access approach to flow-managed persistence in Spring Web Flow. If you apply `@Transactional` to Web services-enabled `@Service` beans, the non-transactional context could be overridden. All pending data changes in the flow-scoped persistence context will be flushed when a read/write transaction specified in the service layer is met in the method-calling chain, leading to so-called "premature flush."

> **Avoid the premature flush**
> If an entity identifier is generated during an insertion (i.e., identity column), even with manual flush, a flush occurs after a call to the `entityManager.persist()` or `entityManager.merge()` method. This is necessary because each managed (persistent) entity in the persistence context must be assigned an identifier. To avoid this premature flush, you need to set the ID-generation strategy to `sequence`.

On the other hand, specifying `@Transactional(readOnly=true)` on your view-tier Spring beans will override the read/write transaction settings on the service beans; the transaction will remain read-only to prevent premature flush. In cases where the entire Web tier is bypassed for SOAP/REST Web services communications, the `@Transactional` annotation applied to the service beans ensures that a Web services invocation runs within a database transaction.

This is a great advantage of using read-only transactions over non-transactional data access in flow-managed persistence.

As previously mentioned, flow-managed persistence resolves use cases involving atomic Web flows. For the remainder of the article we'll focus on use cases that call for non-atomic Web flows, where flow-managed persistence does not apply. Note that in some of these use cases we are still able to use the flow-scoped persistence-context object.

Non-atomic Web flows

From a business process management (BPM) perspective, one kind of long-running process lives longer than the lifespan of a typical Web session. If such a long-running process involves human tasks, then a user can work on the process for any time span and come back in hours, days, even months to restore the execution of the process. Obviously, such a process should survive server crashes, as well.

All of these factors suggest that the state of the long-running process after each progression needs to be persisted to a backend database. Implementing the human activities of this long-running process as a Web flow would be a sound technical solution. The flow would be executed repeatedly in different Web sessions to emulate the lifecycle of the long-running process.

Apart from the above scenario, some applications consist of non-contextual Web pages, which a user can navigate between arbitrarily. Those Web pages could be grouped into flows according to their business functions, even though there would be no logical sequential flows, nor begin or end states. Data changes made during every user request would need to be saved. Persistence programming in these applications is not different from the long-running process discussed above, in that transactional atomicity is scoped to each user action instead of a series of user actions — a Web flow.

A use case for non-atomic Web flows

In the health care industry, service providers periodically approach members who have chronic diseases to evaluate their health status and potential risks. Health providers subsequently offer them medical and behavioral health suggestions. This is known as *case management*.

A case management system is centered on a series of contact tasks. In one typical task, a case manager contacts a member by phone, asks assessment questions and, based on the responses, makes the appropriate suggestions, creates referrals, records the contact outcome, and sets up follow-up tasks.

Complications are many. The list of assessment questions could be long: the call

could be interrupted for various reasons, certain tasks could be unable to complete without referrals being logged, and so on. A contact task contains concurrent or asynchronous operations is a long-running process, and progress at every step needs to be saved to a database. The contact task can be emulated as a single Web flow, which can be entered and executed repeatedly during the course of the long-running process.

This non-atomic Web flow scenario is not covered by the Spring Web Flow documentation. Is it possible to still leverage the flow-scoped persistence-context object in this use case? The answer is yes.

Specifying the scope of transactions

We know that the `<persistence-context/>` tag in a flow-definition file gives us a thread-bound and `flowScoped` persistence context, with the benefits of *no detached entities* and *no* `LazyInitializationException`. So, we choose to keep that tag. Compared with flow-managed persistence in atomic flows, the most significant change happens to the scope of transactions: atomicity applies to each step of the process rather than the entire flow. Quite often, an atomic step in the process is a user action represented by a `<transition>` tag in the Web-flow definition.

It is disappointing that Spring Web Flow doesn't support transaction demarcation on any of its tags, including `<transition>` and `<evaluate>`. The next option for developers is to initiate a database transaction from a Spring bean's method annotated with `@Transactional` and call that method from an `<evaluate>` tag. (The `<transition>` tag doesn't support method invocation.)

In essence, the transaction is scoped to the `<evaluate>` tag in the flow. Applying `@Transactional(readyOnly=false)` will make the JPA/Hibernate `FlushMode` AUTO, so that Hibernate determines when to flush the data changes within the context of the same transaction. For the sake of programming convenience and SQL optimization, auto flush is preferred over manual in these cases. Note that multiple `<evaluate>` tags are allowed under the same `<transition>`, resulting in multiple database transactions per user action.

If each user action/request is considered atomic, which in general is true, we want to group all database write operations inside a single `@Transactional` method of one Spring bean, such that they are bound with the same transaction context and invoked through the same `<evaluate>` tag. Listing 1 shows how we specify the transaction context for an atomic request.

Listing 1. Specifying the transaction context for an atomic user action

```
<transition>
  <evaluate expression="beanA.readAlpha()"/>
  <evaluate expression="beanA.readBeta()"/>
```

```
  <evaluate expression="beanB.readGamma()"/>
  <evaluate expression="beanA.writeAll()"/> <!-- a single read/write transaction -->
  <evaluate expression="beanB.readEta()"/>
</transition>
```

Listing 2 shows an atypical case where multiple read/write transactions (committing or rolling back on their own) are involved in the same user request. Consequently, the user request becomes non-atomic, which in most development scenarios is catastrophic.

Listing 2. Specifying the transaction context for a non-atomic user action

```
<transition>
  <evaluate expression="beanA.readAlpha()"/>
  <evaluate expression="beanA.readBeta()"/>
  <evaluate expression="beanB.readGamma()"/>
  <evaluate expression="beanA.writeDelta()"/> <!-- read/write transaction -->
  <evaluate expression="beanA.writeEpsilon()"/> <!-- read/write transaction -->
  <evaluate expression="beanB.writeZeta()"/> <!-- read/write transaction -->
  <evaluate expression="beanB.readEta()"/>
</transition>
```

How will we handle those read-only operations referenced by other `<evaluate>` tags under the same `<transition>`? We have three options:

1. Run the read-only operations with no database transactions, as discussed previously.
2. Mark them `@Transactional(readOnly=false)` such that the SQL queries are executed under read/write database transactions. In this case, the `FlushMode` of the flow-scoped persistence context will always be `AUTO`.
3. Mark them with `@Transactional(readOnly=true)`. In this case, the `FlushMode` becomes `MANUAL` for these read-only transactions and transitions to `AUTO` once a read/write transaction is reached.

JPA/Hibernate automatically flushes pending changes in the persistence context before a read/write transaction commits. For the sake of simplicity, the Hibernate team encourages developers to apply read/write transactions consistently across all data operations under such circumstances. Just set `readOnly=false` to wherever `@Transactional` applies.

What about one transaction per request?
You may ask why not encapsulate each user request with one big transaction. The problem is that a transaction at the request level would keep the associated database locks open until view-rendering was complete. Unfortunately, view-rendering involves blocking I/O operations, which may be slow enough to impact the performance and scalability of the underlying database.

Unexpected OptimisticLockingFailureException

When using the flow-scoped persistence context in non-atomic Web flows, you may encounter some unexpected `OptimisticLockingFailureExceptions`.

Optimistic locking is still highly recommended for non-atomic Web flows to protect the data integrity of each user action. When the `@Version` field of an entity is a database-generated integer or timestamp, following an update operation, the entity needs to be queried explicitly to refresh its state in the persistence context. If not, the `@Version` field would carry a stale value and subsequent updates on the same entity in different transactions would result in an `OptimisticLockingFailureException`. Ironically, this would happen without multi-user concurrency. In contrast, this query-after-update operation must be avoided in atomic flows, or a premature flush will happen. After all, no matter how many times an entity object is updated in memory during an atomic Web flow, the SQL flush happening at the end of the flow only sees the final state of the entity instance.

It's clear that the flow-scoped persistence context makes persistence programming in both atomic and non-atomic Web flows smoother and simpler. Programming persistence in Web flows without the flow-scoped persistence context object is still feasible but incurs many hurdles and pitfalls.

Persistence programming without the flow-scoped persistence context

In some cases, as the Hotel Booking sample application demonstrates, it is possible to implement a Web flow without the `<persistence-context/>` tag. The most obvious impact of that approach is on atomic Web flows, which are no longer achievable once you've omitted the flow-scoped persistence-context object. I discuss other inconveniences in the sections below.

Persistence context scoped to a database transaction

Without a flow-scoped persistence context, the persistence context injected through the `@PersistenceContext` annotation is by default scoped to a database transaction. To better understand why this is problematic, review the following code snippet from the Hotel Booking sample application:

Listing 3. A fragment of the "main-flow" definition in Hotel Booking

```
<view-state id="enterSearchCriteria">
  <on-render>
    <evaluate expression="bookingService.findBookings(currentUser.name)"
      result="viewScope.bookings" result-type="dataModel" />
```

```
  </on-render>
  <transition on="cancelBooking">
    <evaluate expression="bookingService.cancelBooking(bookings.selectedRow)" />
    <render fragments="bookingsFragment" />
  </transition>
</view-state>
```

If the `cancelBooking` method referenced in Listing 3 is defined as follows:

Listing 4. The cancelBooking method

```
@Service("bookingService")
@Repository
public class JpaBookingService implements BookingService {

//...

@Transactional
public cancelBooking(Booking booking){

  if (booking != null) {
    em.remove(booking);
  }
}
```

Then we'll get the following error when we run the code:

```
Caused by: java.lang.IllegalArgumentException: Removing a detached instance
```

The `booking` entities returned from the `<on-render>` tag become detached in the subsequent action `<transition on="cancelBooking">`. The two methods `findBookings` and `cancelBooking` of the same `bookingService` bean are executed under different database transactions and therefore are associated with two distinct persistence-context objects. The `booking` entities managed by one persistence context are detached from the perspective of the other.

To circumvent this problem, in the actual `cancelBooking` method shown in Listing 5, the same `booking` entity is reloaded through its primary key before it is removed.

Listing 5. The fixed cancelBooking method

```
@Service("bookingService")
@Repository
public class JpaBookingService implements BookingService {

//...

@Transactional
public cancelBooking(Booking booking){

  booking = em.find(Booking.class, booking.getId()); // reinstate the persistent entity

  if (booking != null) {
    em.remove(booking);
  }
```

```
}
```

Effectively, the transaction-scoped persistence context works the same way as `OpenSessionInViewFilter/Interceptor` with `singleSession=false`. This means each transaction in the same request has its own associated session. But here we lose the benefit of Open Session in View's "deferred close mode.".

During view-rendering, lazy reads will lead to `LazyInitializationExceptions`, because the transaction-scoped persistence context is closed immediately after the completion of each transaction. Implementing something similar to `OpenSessionInViewInterceptor` / `OpenEntityManagerInViewInterceptor` is an option, but the ones provided by the Spring Core do not work out of the box for Spring Web Flow. It is much more handy to use the built-in flow-scoped persistence-context object!

Persistence context scoped to each invocation

Non-transactional data access without the assistance of the flow-scoped persistence context is a worst-case scenario and should be avoided, if at all possible.

Outside of a transaction, the persistence context is scoped to each invocation with `FlushMode AUTO` and `auto-commit true`. (Remember that Hibernate disables auto flush for non-transactional data access.) In other words, each method invocation on the same persistence-context proxy injected through `@PersistenceContext` will return a different entity-manager instance, which opens and closes immediately.

Essentially, the entity manager is scoped to a "short transaction." The same code snippet you saw in Listing 5 would then give you the following error message:

```
java.lang.IllegalArgumentException: Removing a detached instance
```

Passing entities in different flows

Here's one last scenario that sometimes causes problems: what happens when you are required to pass entities in different flows?

A flow-scoped persistence-context object is subject to the scope of the flow, therefore, entities once passed from one flow to another immediately become detached. The solution is to either merge/reattach those entities to the persistence context of the current flow or reload them with their primary keys, a strategy that mirrors the Open Session in View approach.

In conclusion

Spring Web Flow as an advanced Web development framework offers unique features to support persistence programming and transaction management with JPA/Hibernate. This article has explored the complexities and challenges Java developers face in programming Spring Web Flow applications.

From real-world use cases like the ones discussed in this article, I have devised the following "rules of thumb" for coding transactional atomic and non-atomic Web applications in Spring Web Flow:

- As a first choice, always use a flow-scoped persistence context
- Apply read-only transactions universally to all methods referenced in an atomic Web flow
- Apply read/write transactions universally to all methods referenced in a non-atomic Web flow

Resources

Learn

- Spring Web Flow Reference Guide: A comprehensive online reference published by the Spring framework team.
- "Hibernate Article: Open Session in View" (Anthony Patricio, August 2009, JBoss.org): Learn more about the OSIV pattern, including using it for non-transactional data access.
- "Seamless JSF, Part 2: Conversations with Seam" (Dan Allen, developerWorks, May 2007): Part of an in-depth introduction to Seam, focusing on Seam conversations.
- "Seam and Spring comparison" (Andy Gibson, Software Development Blog, 2008): A high-level comparison that looks, among other things, at the difference between using Web flows versus conversations.
- "Extended Persistence Context in Stateful Session Beans" (Mahesh Kannan, Sun Developer Network, February 2008): Presents an application where a container-managed entity manager is used with an extended persistence context to handle a long conversation.
- "Spring Web Flow 2: A boon to JSF developers" (Xinyu Liu, JavaWorld, November 2008): A brief introduction of the new features available in the Spring Web Flow 2 release.
- "Use continuations to develop complex Web applications (Abhijit Belapurkar, developerWorks, December 2004): An introduction to Spring MVC's continuations-based programming paradigm.
- Spring Web Flow 2 Web Development: A practical guide to designing powerful Web applications with the Spring Web Flow framework.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Spring Web Flow Project homepage: Get the latest updates to the Spring Web Flow project.

Discuss

- Get involved in the My developerWorks community.

About the author

Xinyu Liu

As a Sun Microsystems certified enterprise architect, Xinyu Liu has intensive application design and development experience on the Java EE, Java SE, and Java ME platforms. He took his graduate degree from George Washington University and currently is a key contributor to the IT department of a healthcare company. Dr. Liu has written for Java.net and JavaWorld.com on topics such as JSF, Spring Security, Hibernate Search, Spring Web Flow, and the Servlet 3.0 specification. He also has worked for Packt Publishing reviewing the books *Spring Web Flow 2 Web Development* and *Grails 1.1 Web Application Development*. This is his first article for IBM developerWorks.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.