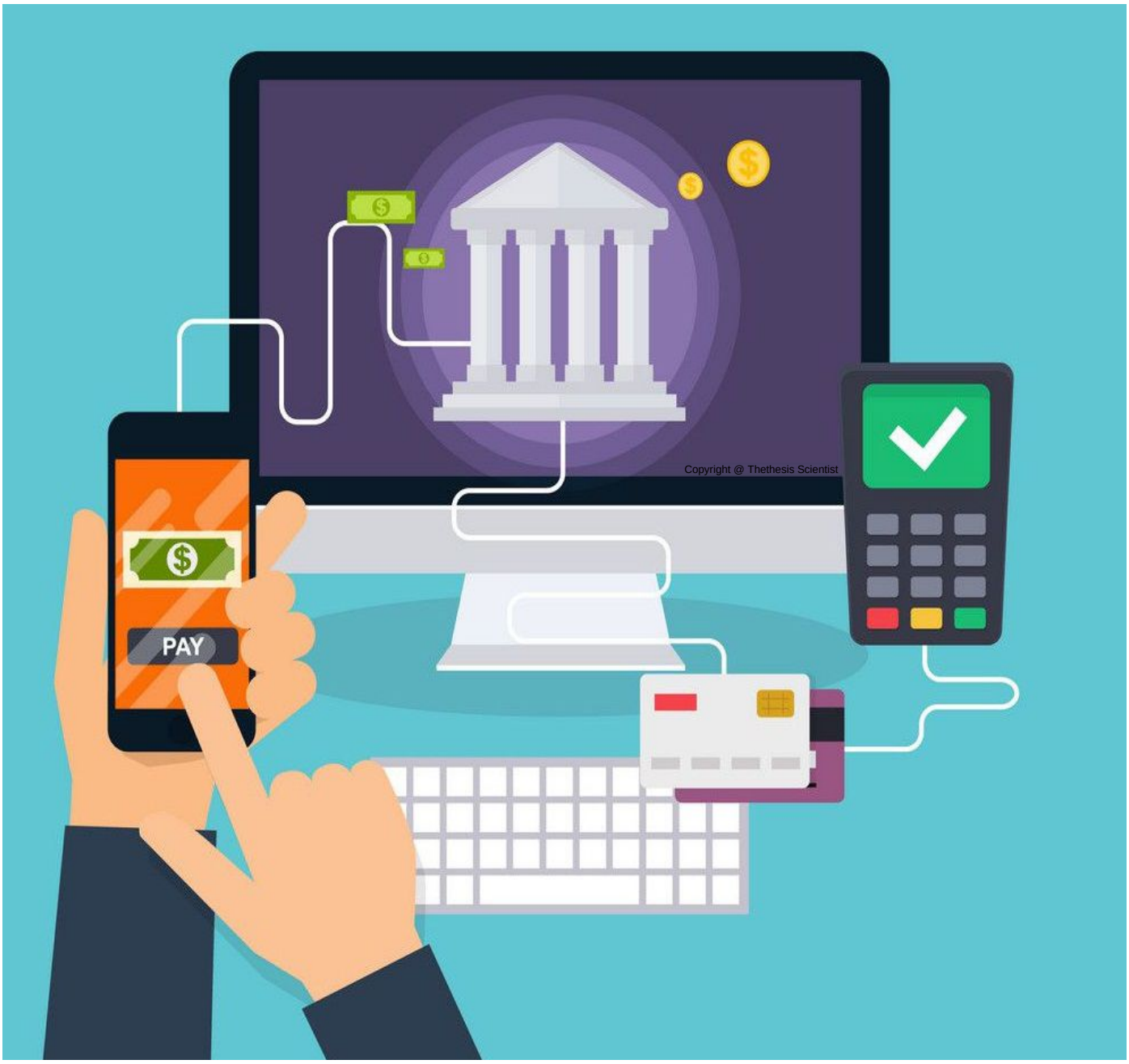
NOTES:

foxmula.com

Java Transaction API (JTA) | Transaction Management



One of the principal benefits of using EJB is its support for enterprise-wide services like transaction management and security control. Here, we will explore how EJB offers Java Transaction API (JTA) and Transaction Management and how you can leverage them to meet your specific requirements.

What Is a Transaction?

In this section, we will explore the following questions:

- What is a transaction, and why is it important to enterprise applications?

- What are the core ACID (atomicity, consistency, isolation, durability) properties that define a robust and reliable transaction?
- What is JTA, what is a distributed transaction, and what is two-phase commit?

A transaction is a group of operations that must be performed as a unit. These operations can be synchronous or asynchronous and can involve persisting data objects, sending e-mail, validating credit cards, and many other events.

A classic example is a banking transfer, in which one operation debits funds from one account (i.e., updates a record in a database table), and another operation credits those same funds to another account (updates another row in that same, or a different, database table).

From the perspective of an external application querying both accounts, there must never be a time when these funds can be seen in both accounts. Nor can a moment exist when the funds can be seen in neither account.

Only when both operations in this transaction have been successfully performed can the changes be visible from another application context. A group of operations that must be performed together in this way, as a unit, is known as a transaction.

The operations in a transaction are performed sequentially or in parallel, typically over a (relatively short) period of time. After they are all performed, the transaction is applied or committed.

If an error or other invalid condition arises during the course of a transaction, the transaction may be canceled, or rolled back, and the operations that had thus far been performed under that transaction context are undone.

Distributed Transactions

When the operations in a transaction are performed across databases or other resources that reside on separate computers or processes, this is known as a distributed transaction.

Such enterprise-wide transactions require special coordination between the resources involved, and they can be extremely difficult to program reliably. This is where JTA comes in, providing the interface that resources can implement and bind to, to participate in a distributed transaction.

The EJB container is a transaction manager that supports JTA, and so it can participate in distributed transactions involving other EJB containers as well as third- party JTA resources, like many database management systems (DBMSs).

This takes the complexity of coordinating distributed transactions off the shoulders of business application developers so that they are free to integrate loosely coupled services and distribute their data across the enterprise however they choose.

In addition, as you will see in the following sections, EJB allows developers to choose whether to demarcate transactions explicitly—with calls to begin, commit, or rollback (cancel) a transaction— or to allow the EJB container to automatically perform transaction demarcation along EJB method boundaries.

The ACID Properties of a Transaction

No, not the electric Kool-Aid kind. Transactions come in all shapes and sizes and can involve synchronous and asynchronous operations, but they all have some core features in common, known as their ACID components.

ACID refers to the four characteristics that define a robust and reliable transaction: atomicity, consistency, isolation, and durability.

Atomicity A transaction is composed of one or more operations that are performed as a group, known as a unit of work. Atomicity ensures that at the conclusion of the transaction, these operations are either all performed successfully (a successful commit), or none of them are performed at all (a successful rollback).

At the end of a transaction, atomicity would be violated if some, but not all, of the operations, completed.

Consistency A consistent transaction has data integrity. Consistency ensures that at the conclusion of the transaction, the data is left in a consistent state, so that database constraint or logical validation rules are not left in violation.

Isolation Transaction isolation specifies that the outside world is not able to see the intermediate state of a transaction. Outside programs viewing the data objects involved in a transaction must not see the modified data objects until after the transaction has been committed.

Transaction isolation is a complex science in itself, and it is largely beyond the scope of this discussion, but suffice it to say that EJB server providers typically offer configurable isolation settings that let you choose the degree to which resources within a transaction's scope can see each other's pending changes.

There is no standard isolation setting, so portable applications should not rely on a particular configuration in their runtime environment.

Durability Transactions that are performed correctly are permanent and cannot be affected by any failure of the system. Committing the data into a relational database, in which the results may

subsequently be queried, typically achieves this requirement.

Naturally, EJB addresses all these requirements, and we will point out how each one is handled in the examples that follow.

The Java Transaction API (JTA)

JTA defines an interface for clients and transaction-aware resource managers to participate in fault-tolerant distributed transactions.

EJB automatically binds to these services, so both clients and enterprise beans can conveniently participate in distributed programming without having to explicitly code logic such as the two-phase commit protocol.

The primary interface for an EJB into its JTA transaction is through the javax.transaction.UserTransaction object interface instantiated by the EJB container and made available either through injection into the enterprise bean class or through a Java Naming and Directory Interface (JNDI) lookup.

The Two-Phase Commit Protocol

If you have programmed logic using relational databases, you may be familiar with the two-phase commit protocol. This strategy gives veto authority to resource managers participating in a distributed transaction, notifying them through a “prepare” command that a commit is about to be issued, and allowing them to declare whether they can apply their changes.

Only if all resource managers indicate through unanimous consent that they are prepared to apply their changes does the final word come down from the transaction manager to actually apply the changes?

Typically, resource managers perform the bulk of the changes during the “prepare” step, so the final “commit” step is trivial to execute.

This reduces the likelihood that errors will occur during the “commit” step. Robust transaction managers and resource managers are capable of handling even this eventuality.

Transaction Support in EJB

This section will explore the following questions:

- What transaction services are available to enterprise beans?

- What transaction services are available to enterprise bean developers?
- How do session beans, message-driven beans (MDBs), and entities interact in a transactional context?

Much of the infrastructure in the EJB server is dedicated to supporting these services, and for good reason. Not only does EJB provide a robust JTA transaction manager, but it also makes it accessible through declarative metadata that can be specified on interoperable, portable business components. Virtually all Java EE applications require transaction services, and EJB brings them to the application developer in a very slick package.

From its inception, the EJB framework has provided a convenient way to manage transactions and access control by letting you define the behavior declaratively on a method-by-method basis.

Beyond these container-provided services, EJB lets you turn control over to your application to define transaction event boundaries and other custom behavior.

In this blog, we look at the role transactions play in accomplishing a range of common tasks in your application. In a typical Java EE application, session beans typically serve to establish the boundaries of a transaction and manipulate entities to interact with the database in the transactional context.

Our examples mix session bean and entity operations to illustrate both the built-in (declarative) and manual behavior provided by EJB. In the spirit of the simplified development model, EJB provides a lot of its most useful features by default, so it should come as no surprise that the default transaction options are both useful and powerful.

EJB Transaction Services

The EJB transaction model is built on this JTA model, in which session beans or other application clients provide the transactional context in which enterprise services are performed as a logical unit of work.

Enterprise services in the Java EE environment that typically operate in a transactional context include creating, retrieving, updating, and deleting entities; sending JMS messages to the queue; executing MDBs; firing e-mail requests; invoking Web services; executing JDBC operations; and much more.

EJB provides a built-in JTA transaction manager, but the real power lies in the declarative services that EJB offers to bean providers.

Using metadata tags instead of programmatic logic, EJB developers can seamlessly participate in JTA transactions and declaratively control the transactional behavior of each business method on an enterprise bean.

EJB extends this programming model by providing explicit support for both JTA transactions and non-JTA (resource-local) transactions. Resource-local transactions are restricted to a single resource manager, such as a database connection, but may result in a performance optimization by avoiding the overhead of a distributed transaction monitor.

In addition, application builders may leverage the container-provided services for automatically managing transactions, or they may choose to take control of the transaction boundaries and handle the transaction begin, commit, and rollback events explicitly. Within a single application, both approaches may be used, in combination if desired.

Whereas the choice of whether to have the container or the application itself demarcate transactions is defined on the enterprise bean, the decision of which type of transaction model to use with JPA entities—JTA or resource-local—is determined by how the persistence unit is configured in the persistence.xml file.

The persistent objects in the game—the entities—are entirely, and happily, unaware of their governing transaction framework.

The transactional context in which an entity operates is not part of its definition, and so the same entity class may be used in whatever transactional context the application chooses, provided an appropriate EntityManager is created to service the entity's lifecycle events.

If all of this seems a little daunting at this point, fear not. It will all make sense once we walk through some examples in code that demonstrate how all the pieces work together.

It is also worth noting that EJB's built-in Container-Managed Transaction (CMT) support is more than adequate for most applications running inside the EJB container.

This blog will let you explore your options—but unless you are writing applications involving entities that live wholly outside of the EJB container, the default support is likely to serve your needs quite nicely.

Session Bean Transactional Behavior in the Service Model

This next section will explore the following questions:

This next section will explore the following questions:

- **What declarative transaction support does the EJB container offer to session beans?**
- What is the difference between container-managed transaction (CMT) demarcation and bean-managed transaction (BMT) demarcation? When would you choose one approach over the other?
- **What are CMT attributes?**
- How does EJB support the explicit demarcation of transaction boundaries using bean-managed transaction (BMT) beans?
- **What is implicit vs. explicit commit behavior?**

The enterprise bean is the heart of the EJB service layer. Through session beans, the EJB container offers declarative demarcation of transaction events, along with the option to demarcate transaction events explicitly in the bean or in the application client code.

Let’s consider these two approaches separately, beginning with the default option: leveraging container-managed transaction demarcation using declarative markup.

Container-Managed Transaction (CMT) Demarcation

The EJB container provides built-in transaction management services that are available by default to session beans and MDBs.

The bean designates transaction characteristics for each of its methods through metadata (using either annotations or XML), and the EJB container follows those directives to determine the transactional action (if any) to perform.

Using these directives, the EJB container may automatically begin a new transaction or suspend or reuse an existing transaction when the method is invoked, and possibly commit the transaction before the method returns to the caller.

Note Only the bean developer may assign the transaction manager type of an enterprise bean. Application assemblers are permitted to override the bean's method-level container-managed transaction attributes through the ejb-jar.xml file, but they must use caution to avoid the possibility of deadlock.

Application assemblers and deployers are generally not permitted to override the bean's transaction management type.

When an EJB declares its transactional behavior in metadata, the container interposes on calls to the enterprise bean's methods and applies transactional behavior at the session bean's method boundaries, providing a fixed set of options that you can specify for each method.

The default behavior provided by the container (i.e., when no other directive is specified) is to check, immediately before invoking the method, whether a transaction context is associated with the current thread.

If no transaction context is available, the container begins a new transaction before executing the method. If a transaction is available, the container allows that transaction to be propagated to the method call and made available to the method code. Then, upon returning from the method invocation, the container checks again.

If the container was responsible for creating a new transaction context, it automatically commits that transaction after the method is exited. If it didn't create the transaction, then it allows the transaction to continue unaffected.

By interposing on the bean's method calls, the EJB container is able to apply transactional behavior at runtime that was specified declaratively at development time.

The default behavior, described in the previous paragraph, is but one of six CMT demarcation options provided by the container. You can attribute any one of these six demarcation options to any method on a session bean.

Some of the attribute values require specific conditions to be met; when they are not met, an exception is thrown.

Container Transaction Attribute Definitions

Transaction Attribute Behavior

MANDATORY A transaction must be in effect at the time the method is called. If no transaction is available, a javax.ejb.The ejbtransactionrequired exception is thrown. This transaction remains in effect while the method is executed, and it is left active upon returning control to the caller.

REQUIRED This is the default transaction attribute value. Upon entering the method, the container interposes to create a new transaction context if one is not already available.

If the container created a transaction upon entering the method, it commits that transaction when the method call completes. If a transaction was already in effect, the container does not commit it before returning control to the client.

REQUIRES_NEW The container always creates a new transaction before executing a method thusly marked. If a transaction context is already available when the method is invoked, then the container suspends that transaction by dissociating it from the current thread before creating the new transaction.

The container then reassociates the original transaction with the current thread after committing the intervening one.

SUPPORTS This option is basically a no-op, resulting in no additional work by the container. If a transaction context is available, it is used by the method.

If no transaction context is available, then the container invokes the method with no transaction context. Upon exiting the method, any preexisting transaction context remains in effect.

NOT_SUPPORTED The container invokes the method with an unspecified transaction context.

If a transaction context is available when the method is called, then the container dissociates the transaction from the current thread before invoking the method, and then it reassociates the transaction with the thread upon returning from the method.

NEVER The method must not be invoked with a transaction context. The container will not create one before calling the method, and if one is already in effect, the container throws javax.ejb.EJBException.

Note In general, transaction attributes may be specified on a session bean's business interface methods or Web service endpoint interface, or on an MDB's listener method, but some additional restrictions apply for specific cases. Such details are beyond the scope of this blog but may be found in the EJB Core Contracts and Requirements spec.

All six attributes are typically available for session bean methods, though certain attributes are not available on a session timeout callback method, or when the session bean implements javax.ejb.SessionSynchronization.

MDBs support only the REQUIRED and NOT_SUPPORTED attributes. Here is an example of how you would specify the transaction behavior on a session bean method to override the transaction behavior specified (or defaulted) at the bean level:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public CustomerOrder createCustomerOrderUsingSupports(Customer customer) throws Exception { ... }
The EJBContext.setRollbackOnly and getRollbackOnly Methods
```

In the event that an exception or other error condition is encountered in a method on a CMT-demarcated enterprise bean, the bean may wish to prevent the context transaction from being committed.

The MessageDrivenContext methods can be used, in the following ways, whenever we use container-managed transactions:

- setRollbackOnly: This method can be used for error handling.
- getRollbackOnly: This method can be used to test whether the current transaction has been marked for rollback.

The bean is not allowed to roll back the transaction explicitly, but it may obtain the javax.ejb.EJBContext resource (through container injection or JNDI lookup) and call its setRollbackOnly() method to ensure that the container will not commit the transaction.

Similarly, a bean method may at any time call the EJBContext. getRollbackOnly() method to determine whether the current transaction has been marked for rollback, whether by the current bean or by another bean or resource associated with the current transaction.

Bean-Managed Transaction (BMT) Demarcation

For some enterprise beans, the declarative CMT services may not provide the demarcation granularity that they require. For instance, a client may wish to call multiple methods on a session bean without having each method commit its work upon completion.

In this case, the client has several options: it can either instantiate its own JTA (or resource-local) transaction and thus control the transaction begin/end boundaries explicitly;

It can write a custom CMT session bean to wrap the work inside a transactional bean method and perform the steps inside the container-managed transaction, or it can control the transaction demarcation explicitly by using a transaction resource available through the EJB context.

EJB offers the latter option—known as bean-managed transaction (BMT) support to enterprise beans as a convenient way to handle their demarcation of transaction events.

To turn off the automatic CMT demarcation services, enterprise beans simply specify the @TransactionManagement annotation or assign the equivalent metadata to the session bean in the ejb-jar.xml file.

With BMT demarcation, the EJB container still provides the transaction support to the bean, through a UserTransaction object available through the bean's EJBContext object.

The primary difference is that the bean code makes explicit calls to begin, commit, and roll back transactions instead of using CMT attributes to declaratively assign transactional behavior to its methods.

The container does not interpose on BMT methods to begin and commit transactions, and it does not propagate transactions begun by a client to beans that elect to demarcate their own transactions.

While any given enterprise bean must choose one plan or the other (CMT vs. BMT demarcation) for its methods, both types of beans may interact with each other within a single transaction context.

To demarcate transactions, an enterprise bean obtains an EJBContext (that is, SessionContext for a session bean, or MessageDrivenContext for an MDB) resource through injection:

```
@Resource
SessionContext sessionContext;
and then acquires a JTA javax.transaction.UserTransaction instance through this resource:
UserTransaction txn = sessionContext.getUserTransaction();
```

This EJB container-provided UserTransaction interface provides begin(), commit(), and rollback() transaction demarcation methods to the bean. Similarly, non–enterprise bean clients may acquire a UserTransaction resource from a JTA server other than the EJB container to demarcate transactions from an application client environment.

or they may use a non-JTA resource-local EntityTransaction obtained through an EntityManager (see below). Regardless of how they begin a transaction, though, it becomes the transaction in context when they invoke a session bean method.

In the example we will examine shortly, a stateful session bean using an EXTENDED persistence context and BMT demarcation initiates a transaction in one method that is then propagated to subsequent method calls until the transaction is finally committed in a separate method.

This behavior could not be specified with the same method structure by calling these methods separately on a CMT session bean, although it would be possible to achieve the same results by wrapping these separate calls inside a single CMT-demarcated custom method on a CMT session bean.

Note How does the EJB server wire up a transaction context? You may be curious as to how the EJB server is able to automatically enlist database connections, and other resources obtained programmatically inside an enterprise bean, with the transaction context.

Since the EJB server is providing the context for executing bean methods, it is able to interpose on these requests and perform this side-effect logic without interrupting the flow of execution within the method.

That is, it intercepts the method invocation before it is performed; does some extra work (like checking the state of the transaction context, possibly creating a new transaction, and associating the enterprise bean with that context); and then invokes the bean method.

Upon returning from the bean method invocation, it again has an opportunity to perform extra logic, such as committing the transaction that was created when the method was called, before returning control to the client that invoked the bean method in the first place.

When we use bean-managed transactions, the delivery of a message to the onMessage method takes place outside the JTA transaction context. The transaction will begin when we call the UserTransaction.begin method within the onMessage method, and end when we call UserTransaction.commit or UserTransaction. rollback.

Notice that any call to the Connection.createSession method should take place within the transaction.

Implicit Commit vs. Explicit Commit

When using CMT, the container requires that any transaction that it begins when a method is called will be concluded when the method returns to the caller. This is known as implicit commit behavior or sometimes referred to as auto-commit behavior.

The container must enforce this rule or run the risk of losing track of the transaction as control passes up the process stack.

While you can still use the container-managed transactions with CMT to allow multiple methods to be called within a single transaction, you must do this by wrapping your method calls within an outer container-managed method. From the perspective of a client, this still appears to be a single method call, to the wrapper method.

When using BMT with stateful session beans, you have the flexibility to achieve an explicit commit model, where the bean or the client can control the beginning and end of a transaction and call multiple methods explicitly during that time before concluding with an explicit call to commit or roll back the work.

This puts the burden on the client, or on the bean developer, to ensure that transactions are not left

dangling. With proper care, however, this is a powerful tool when the application requires this behavior.

As mentioned earlier, EJB provides a built-in transaction for session beans and MDBs to use for this—the javax.transaction.UserTransaction—which the bean can access through its injected javax.ejb.EJBContext property (as either a SessionContext or a MessageDrivenContext instance).

We will further explore both implicit and explicit commit behavior in the sample app at the end of this blog.

Using Transactions with JPA Entities

This section will discuss the following questions:

- How are transactions managed in the persistence layer?
- What options does the persistence framework offer for controlling transactions involving entities?
- What is the role of the persistence context in a transaction?
- How do entities become associated with, and dissociated from, a transactional context?

If you recall from the Previous Blog, a persistence unit defines a set of entity classes, and a persistence context is a managed set of entity instances from a single persistence unit.

At any point in time, across multiple applications executing in an application server, many persistence contexts may be actively associated with any given persistence unit, but each persistence context is associated with, at most, one transaction context.

Relationship Between Entities and a Transaction Context

From the preceding discussion about how the EJB server acts as a transaction coordinator in associating resources with a transaction context, you may have realized that the persistence context is the resource that gets associated with a transaction.

In this way, a persistence context is propagated through method calls so that entities in a persistence unit can see each other's intermediate state, through their common persistence context, whenever they are associated with the same transaction context.

Also, the restriction that only one persistence context for any given persistence unit must be associated with a given transaction context ensures that for any entity of type T with identity I, its state will be represented by only one persistence context within any transaction context.

Within an application thread, only one transaction context is available at any moment, but the EJB server is free to dissociate one persistence context from that thread and associate a new persistence context for the same persistence unit to satisfy transaction isolation boundaries.

When the EJB server does this, the newly instantiated persistence context is not able to see the intermediate changes made to any entities associated with the suspended persistence context.

Container-Managed vs. Application-Managed Persistence Context

The persistence services in EJB let you opt out of container-managed persistence contexts altogether and manage the life cycles of your persistence context explicitly within your application code. When an EntityManager instance is injected (or acquired through JNDI), it comes in as a container-managed persistence context.

The container automatically associates container-managed persistence contexts with any transaction that happens to be in context at the time that the EntityManager is used, and it destroys the persistence context when the transaction concludes (with one caveat—see extended persistence contexts, below).

Should an application wish to control how or whether its persistence contexts are associated with transactions, and whether it survives past transactional boundaries, it may obtain an EntityManagerFactory (again, through container injection or JNDI lookup) and explicitly create the EntityManager instances that manage their persistence contexts?

An application-managed persistence context is used when the EntityManager is obtained through an EntityManagerFactory—a requirement when running outside the Java EE container.

For more information on using an application-managed EntityManager outside of a Java EE container, as in a pure Java SE environment.

Transaction-Scoped Persistence Context vs. Extended Persistence Context

A persistence context that is created when a transaction is created and destroyed when the transaction ends, is known as a transaction-scoped persistence context. This is the behavior of persistence contexts associated with all EntityManagers used on a stateless session bean.

For stateful session beans, there exists a special form of container-managed EntityManager that is not bound to the life of a transaction, but it is instead bound to the life of a stateful session bean itself.

This is known as an extended EntityManager, and it behaves much like an application-managed persistence context but the EJB container conveniently manages its life cycle.

Because an extended persistence context is not destroyed at the conclusion of each transaction as is a transaction-scoped persistence context, entities can remain in a managed state even after they have been synchronized with the database.

In some cases, this avoids the need to re-query them or otherwise obtain a managed instance if you wish to continue working with them after a commit. In a conversational environment, such as a Web application, this can be very useful.

An extended persistence context stays open until its context stateful session bean is destroyed. Only stateful session beans may use extended persistence contexts. At the time an EntityManager instance

is created, its persistence context type is defined, and it may not be changed during the EntityManagers lifetime.

The default type is transaction scoped; to inject an EntityManager by specifying an extended persistence context, you may specify the injection directive with the following:

```
@PersistenceContext(unitName="WineAppUnit", type = PersistenceContextType. EXTENDED) private EntityManager em; or you may define a persisten ce-context-ref element in the XML descriptor.
```

In the transaction examples at the end of this blog, we will compare the behavior of a stateless session bean using a transaction-scoped persistence context with a stateful session bean that uses an extended persistence context.

JTA vs. Resource-Local EntityManagers

An EntityManager may be defined to participate in either a JTA transaction or a non-JTA (resource-local) transaction. The features of JTA—most notably, support for distributed transactions—have been described previously, along with usage of the bean's interface to a JTA transaction,

javax.transaction.UserTransaction.

> Resource-local EntityManagers service transactions using the javax.persistence.EntityTransaction interface available to clients through the EntityManager .getEntityTransaction() method.

This interface exposes the expected transaction demarcation methods begin(), commit(), and rollback(), along with getRollbackOnly() and setRollbackOnly() methods that are equivalent to the EJBContext and UserTransaction methods available to enterprise beans described previously, and an isActive() method to indicate whether a transaction is currently in progress.

Container-managed EntityManagers must be JTA EntityManagers.

Application- managed EntityManagers may be either JTA or resource-local, but they may only be JTA EntityManagers if the EntityManager resides in the Java EE environment.

One reason you might want to use a resource-local EntityManager is that while JTA provides the infrastructure for distributed transactions, resource-local transactions can provide a performance optimization by eliminating the overhead of this infrastructure.

Another reason is you may wish to use your JPA entities in a stand-alone Java SE environment where JTA or/or data-source resources are not supported.

We will examine the use of the EntityTransaction when we dissect a Java facade in the sample application later in this blog.

Stateless Session Beans with CMT Demarcation

We begin with a default, straightforward, implementation of a stateless session bean, OrderProcessorCMTBean.java.

This session bean uses CMT demarcation to leverage EJB's declarative transaction support. It is followed by a simple servlet client, OrderProcessorCMTClient.java.

```
OrderProcessorCMTBean.java, a Stateless Session Bean Using CMT Demarcation
@Stateless(name = "OrderProcessorCMT", mappedName = "Blog08- TransactionSamples-OrderProcessorCMT") public class OrderProcessorCMTBean
{
@Resource
SessionContext sessionContext;
@PersistenceContext(unitName = "Blog08-TransactionSamples-JTA") private EntityManager em;
/**
* Remove any existing Customers with email 'wineapp@yahoo.com' and any existing Wine with
* country 'United States'. The EJB container will ensure that this work is performed in
* a transactional context.
*/
public String initialize() {
StringBuffer strBuf = new StringBuffer();
strBuf.append("Removed ");
int i = 0;
// Filter the data by removing any existing Customers with email
// 'wineapp@yahoo.com' (or whatever is defined in the user.properties file). for (Customer customer :
getCustomerFindByEmail(PopulateDemoData.TO_EMAIL_ADDRESS)) { em.remove(customer);
i++;
}
strBuf.append(i);
strBuf.append(" Customer(s) and ");
// Remove any existing Wine with country 'United States' i = 0;
for (Wine wine : getWineFindByCountry("United States")) { em.remove(wine);
i++;
}
strBuf.append(i); strBuf.append(" Wine(s)"); return strBuf.toString();
}
/**
* Create a new CustomerOrder from the items in a Customer's cart. Creates a new CustomerOrder
* entity, and then creates a new OrderItem entity for each CartItem found in the Customer's cart.
*
* Using CMT w/ the default REQUIRED xaction attribute, if this method is invoked without a
* transaction context, a new transaction will be created by the EJB container upon invoking the
* method, and committed upon successfully completing the method.
*
* @return a status message (plain text)
*/
public CustomerOrder createCustomerOrder(Customer customer) { return createCustomerOrderUsingSupports(customer);
}
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public CustomerOrder createCustomerOrderUsingSupports(Customer customer)
{
if (customer == null) {
throw new IllegalArgumentException("OrderProcessingBean.
createCustomerOrder(): Customer not specified");
}
if (!em.contains(customer)) {
customer = em.merge(customer);
}
```

```
final CustomerOrder customerOrder = new CustomerOrder(); customer.addCustomerOrder(customerOrder);
final Timestamp orderDate = new Timestamp(System.currentTimeMillis()); final List<CartItem> cartItemList =
new ArrayList(customer.getCartItemList()); for (CartItem cartItem : cartItemList) {
// Create a new OrderItem for this CartItem final OrderItem orderItem = new OrderItem(); orderItem.setOrderDate(orderDate); orderItem.setPrice(cartItem.
getWine().getRetailPrice()); orderItem.setQuantity(cartItem.getQuantity()); orderItem.setStatus("Order Created"); orderItem.setWine(cartItem.getWine()); c
ustomerOrder.addOrderItem(orderItem);
// Remove the CartItem customer.removeCartItem(cartItem);
}
return persistEntity(customerOrder);
}
public <T> T persistEntity(T entity) { em.persist(entity);
return entity;
}
public <T> T mergeEntity(T entity) {
return em.merge(entity);
}
public <T> void removeEntity(T entity) { em.remove(em.merge(entity));
}
public <T> List<T> findAll(Class<T> entityClass) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery(); cq.select(cq.from(entityClass));
return em.createQuery(cq).getResultList();
}
public <T> List<T> findAllByRange(Class<T> entityClass, int[] range) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery(); cq.select(cq.from(entityC
lass));
Query q = em.createQuery(cq);
q.setMaxResults(range[1] - range[0]);
q.setFirstResult(range[0]);
return q.getResultList();
}
/**
* <code>select o from Customer o where o.email = :email</code>
*/
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED) public List<Customer> getCustomerFindByEmail(String email) {
return em.createNamedQuery("Customer.findByEmail", Customer.class). setParameter("email", email).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.country = :country</code>
*/ @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED) public List<Wine> getWineFindByCountry(String country) {
return em.createNamedQuery("Wine.findByCountry", Wine.class). setParameter("country", country).getResultList();
}
}
```

Listing OrderProcessorCMTClient.java, a Servlet That Drives the OrderProcessorCMT Session Bean

```
@WebServlet(name = "OrderProcessorCMTClient", urlPatterns = {"/ OrderProcessorCMTClient"})
public class OrderProcessorCMTClient extends HttpServlet { @EJB
OrderProcessorCMTBean orderProcessorCMT;
/**
* Processes requests for both HTTP
* <code>GET</code> and
* <code>POST</code> methods.
*
* @param request servlet request
* @param response servlet response
* @throws ServletException if a servlet-specific error occurs
* @throws IOException if an I/O error occurs
*/
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException { response.setContentType("text/html;charset=UTF-8"); response.setContentType("text/html;charset=UTF-8"); Out
putStream rOut = response.getOutputStream(); PrintStream out = new PrintStream(rOut);
try {
/* TODO output your page here. You may use following sample code. */
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet OrderProcessorCMTClient</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Servlet OrderProcessorCMTClient at " + request.
getContextPath() + "</h1>");
out.println("</body>");
out.println("</html>");
```

```
// Create and persist a bunch of JPA entities, populating the database with data
out.print("<h2>Populating Demo Data... ");
PopulateDemoData.resetData("Blog07-WineAppUnit-ResourceLocal",
System.out);
out.println("done</h2>");
// Filter the data by removing any existing Customers with email
// 'wineapp@yahoo.com' (or whatever is defined in the user. properties file).
// The first call to a transactional method on OrderProcessorBMT will begin a
// transaction.
out.print("<h2>Filtering Demo Data... "); System.out.println(orderProcessorCMT.initialize()); out.println("done</h2>");
// Create a Customer and add some CartItems and their associated Wines Individual customer = new Individual(); customer.setFirstName("Transaction");
customer.setLastName("Head"); customer.setEmail(PopulateDemoData.TO_EMAIL_ADDRESS);
for (int i = 0; i < 5; i++) { final Wine wine = new Wine(); wine.setCountry("United States"); wine.setDescription("Delicious wine"); wine.setName("Xacti"); win
e.setRegion("Dry Creek Valley"); wine.setRetailPrice(new Float(20.00D + i)); wine.setVarietal("Zinfandel"); wine.setYear(2000 + i); orderProcessorCMT.p
ersistEntity(wine);
final CartItem cartItem = new CartItem(); cartItem.setCreatedDate(new Timestamp(System.currentTimeMillis()));
cartItem.setCustomer(customer);
cartItem.setQuantity(12);
cartItem.setWine(wine);
customer.addCartItem(cartItem);
}
// Persist the Customer, relying on the cascade settings to persist all
// related Wine and CartItem entities as well. After the call, the Customer
// instance will have an ID value that was assigned by the EJB container
// when it was persisted.
orderProcessorCMT.persistEntity(customer);
// Create a customer order and create OrderItems from the CartItems final CustomerOrder customerOrder =
orderProcessorCMT.createCustomerOrder(customer);
out.print("<h2>Retrieving Customer Order Items... ");
for (OrderItem orderItem: customerOrder.getOrderItemList()) { final Wine wine = orderItem.getWine(); out.println(wine.getName() + " with ID " + wine.getId
());
}
out.println("done</h2>");
} finally { rOut.close();
out.close();
}
}
/* HttpServlet Methods */
}
```

Transaction Analysis

The following sections will analyze this test run from a transaction perspective. Populating Test Data Through a Transactional Java facade.

The servlet client begins by wiping the slate clean. The client of a CMT bean does not create transactions, or otherwise, concern itself with transactional details. It delegates all work to the CMT session bean and (indirectly to) a Java facade and relies on them to perform their work in a transactional context.

The client first calls the PopulateDemoData class from sample application, which is a helper class that delegates to a Java facade that uses an application-managed EntityManager, to reset the demo data to prepare for a new test run.

We offer this example first, to see a raw use of transactions in an application-managed EntityManager context. We will examine EJB session bean behavior in a moment.

```
 style="margin:0;width:991px;height:84px">// Create and persist a bunch of JPA entities, populating the database with data
out.print("<h2>Populating Demo Data... ");
PopulateDemoData.resetData("WineAppUnit-ResourceLocal", System.out);
out.println("done</h2>");
The PopulateDemoData class is shown in Listing:
```

Listing PopulateDemoData.java, a utility class resets the sample data by delegating to a transactional Java facade over JPA entities

```
public class PopulateDemoData {
public static final String FROM_EMAIL_ADDRESS; public static final String TO_EMAIL_ADDRESS;
static {
Properties properties = new Properties(); InputStream is = null; try {
is = PopulateDemoData.class.getClassLoader().getResourceAsStream
("user.properties");
properties.load(is);
FROM_EMAIL_ADDRESS = properties.getProperty("from_email_address"); TO_EMAIL_ADDRESS = properties.getProperty("to_email_address");
} catch (IOException e) { throw new RuntimeException(e);
} finally {
if (is != null) { try {
is.close();
} catch (IOException ex) {
Logger.getLogger(PopulateDemoData.class.getName()).log(Level. SEVERE, null, ex);
}
}
}
}
private JavaServiceFacade facade;
public static void main(String[] args) { PopulateDemoData.resetData("Blog07-WineAppUnit-ResourceLocal", System.out);
}
public static void resetData(String persistenceUnit, PrintStream out) { PopulateDemoData pdd = null;
try {
pdd = new PopulateDemoData(persistenceUnit);
out.println("Reporting existing data...
```

```
out.println("Reporting existing data..."); pdd.showDataCount(out);
out.println("Removing data...");
pdd.removeAllDemoData(out);
out.println("Reporting data after removal..."); pdd.showDataCount(out);
out.println("Populating data...");
pdd.populateDemoCustomer();
pdd.populateWines();
out.println("Reporting final data...");
pdd.showDataCount(out);
} finally {
if (pdd != null) { pdd.releaseEntityManager();
}
}
}
private PopulateDemoData(String persistenceUnit) { facade = new JavaServiceFacade(persistenceUnit);
}
private void removeAllDemoData(PrintStream out) { removeAll(OrderItem.class, out); removeAll(CustomerOrder.class, out); removeAll(Individual.class, o
ut); removeAll(Distributor.class, out); removeAll(Supplier.class, out); removeAll(InventoryItem.class, out); removeAll(CartItem.class, out); removeAll(Wine.
class, out);
}
private <T> void removeAll(Class<T> entityClass, PrintStream out) { int i = 0;
for (T entity : facade.findAll(entityClass)) { facade.removeEntity(entity);
}
out.println("Removed " + i + " " + entityClass.getSimpleName() + " instances");
}
private Customer populateDemoCustomer() {
Address a = new Address("Redwood Shores", "CA", "200 Oracle Pkwy", null, "94065");
Individual i = new Individual("James", "Brown", "800.888.8000", TO_EMAIL_ADDRESS, a, a, "04/14", "123"); facade.persistEntity(i);
return i;
}
private InventoryItem populateWines() { InventoryItem ii = null;
for (int i = 0; i < 6; i++) {
Wine w = new Wine("USA", "Fine Wine - ranked #" + i, "Yerba Buena " + i, 90, "Napa Valley", new Float(10 + i), "Zinfandel", 2000 + i); facade.persistEntity(
w);
ii = new InventoryItem(10 + i, w, new java.util.Date(System. currentTimeMillis()), new Float(1 + i)); facade.persistEntity(ii);
}
for (int i = 4; i < 10; i++) {
Wine w = new Wine("France", "Fine Wine - ranked #" + i, "Chateau
Brown " + i, 90, "Loire Valley ", new Float(10 + i), "Zinfandel",
2000 + i);
facade.persistEntity(w);
ii = new InventoryItem(10 + i, w, new java.util.Date(System. currentTimeMillis()), new Float(1 + i)); facade.persistEntity(ii);
}
return ii;
}
public void showDataCount(PrintStream out) {
out.println(facade.getCount(Address.class) + " Addresses found");
out.println(facade.getCount(BusinessContact.class) + " Business
Contacts found");
out.println(facade.getCount(CustomerOrder.class) + " Customer Orders found");
out.println(facade.getCount(Wine.class) + " Wines found");
out.println(facade.getCount(WineItem.class) + " Wine Items found");
}
private void releaseEntityManager() {
if (facade != null) {
facade.close();
}
}
}
```

Notice that instead of injecting an EJB facade using the @EJB notation.

we instantiate the Java facade (JavaServiceFacade) through its constructor and pass it the name of a RESOURCE_LOCAL persistence unit, which we can see from the original call is "Blog07-WineAppUnit-ResourceLocal".

This helper class is calling operations like persistEntity() and removeCustomerOrder() on the facade without explicitly calling for the operation to be committed, which is an indication that the facade uses

implicit commit behavior.

We also take care to ensure that the facade is notified, through its own close() method, after it is used.

This allows it to release its own resources: notably, its EntityManagerFactory and EntityManager resources. These are housekeeping items you don’t need to worry about with EJBs since the EJB container handles this level of resource management for you.

Java Facade Using Application-Managed EntityManager

This brings us to the details of the Java facade itself. This facade was actually quietly introduced in Blog, where it is bundled with the common persistence archive we are sharing from that blog. We will now see how this is handled, by now examining the JavaServiceFacade class.

```
JavaServiceFacade.java, a transactional Java facade over JPA entities, exhibiting implicit commit behavior
public class JavaServiceFacade {
private final EntityManagerFactory emf;
private final EntityManager em;
public JavaServiceFacade() { this("Blog13-EmbeddableEJBTests-ResourceLocal");
}
public JavaServiceFacade(String persistenceUnit) {
emf = Persistence.createEntityManagerFactory(persistenceUnit); em = emf.createEntityManager();
}
public void close() {
if (em != null && em.isOpen()) {
em.close();
}
if (emf != null && emf.isOpen()) {
emf.close();
}
}
/**
* All changes that have been made to the managed entities in the persistence context are applied
* to the database and committed.
*/
private void commitTransaction() {
final EntityTransaction entityTransaction = em.getTransaction(); if (!entityTransaction.isActive()) {
entityTransaction.begin();
}
entityTransaction.commit();
}
public <T> T persistEntity(T entity) { em.persist(entity); commitTransaction();
return entity;
}
public <T> T mergeEntity(T entity) { entity = em.merge(entity); commitTransaction();
return entity;
}
public <T> void removeEntity(T entity) { em.remove(em.merge(entity)); commitTransaction();
}
public <T> List<T> findAll(Class<T> entityClass) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery(); cq.select(cq.from(entityClass));
return em.createQuery(cq).getResultList();
}
public <T> int getCount(Class<T> entityClass) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery(); Root<T> rt = cq.from(entityClass); cq.select(em
.getCriteriaBuilder().count(rt)); javax.persistence.Query q = em.createQuery(cq);
return ((Long) q.getSingleResult()).intValue();
}
/**
* <code>select object(wine) from Wine wine where wine.year = :year</code>
*/
public List<Wine> getWineFindByYear(int year) {
return em.createNamedQuery("Wine.findByYear", Wine.class).
setParameter("year", year).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.country = :country</code>
*/
public List<Wine> getWineFindByCountry(String country) { return em.createNamedQuery("Wine.findByCountry", Wine.class). setParameter("country", co
untry).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.varietal = :varietal</code>
*/
public List<Wine> getWineFindByVarietal(String varietal) {
return em.createNamedQuery("Wine.findByVarietal", Wine.class). setParameter("varietal", varietal).getResultList();
}
}
```

This facade obtains its EntityManager from an EntityManagerFactory, and so the life cycle of this EntityManager is now the responsibility of the facade is now the responsibility of the facade instead of the container.

An application-managed EntityManager is bound to a persistence context cache that can exist outside of a transactional context, and live through multiple transactions, which is essentially the same as an EXTENDED persistence context for a stateful session bean. This allows calls such as EntityManager.persist() to be made before a transaction has been started

EntityManager.persist() to be made before a transaction has been started.

Such an out-of-transaction call would add a new entity to the persistence context but would not immediately result in an SQL call to update the database.

Note Even before an EntityManager transaction is begun, the persistence context quietly begins its own private transaction, as needed, to service any ID generator requests when EntityManager.persist( ) is called.

Because we have bound the PK fields of our entities to a @GeneratedValue ID generator, these IDs are actually eagerly obtained and assigned to the entities during the persist( ).

The persistence context cache is not flushed to the database until the transaction is actually begun, through an EntityTransaction.begin() call, which could be immediately before an EntityTransaction.commit() call is performed.

That is, you have the choice to begin the transaction before any changes are applied to the persistence context or defer the beginning of the transaction until you are ready to commit. In our sample, we defer the begin() call to inside the commitTransaction() method:

```
private void commitTransaction() {
final EntityTransaction entityTransaction = em.getTransaction(); if (!entityTransaction.isActive()) {
entityTransaction.begin();
}
entityTransaction.commit();
}
```

Contrast this behavior with a stateless session bean using the default CMT behavior of TransactionAttributeType.REQUIRED, which implicitly begins a transaction when a session bean method call is made, and it commits the work to the database upon returning from that method call.

The transaction begins and ends within the boundaries of that method call, and such CMT beans would never expose commitTransaction() or rollbackTransaction() methods to their clients, as we will see in the BMT example next.

The implicit commit behavior avoids leaving uncommitted data hanging around in a cache, vulnerable to lose due to hardware or network failure.

However, it also incurs additional back-end processing to complete the work and commits it each time an atomic operation is performed (not to mention, a persistence context cache is created and destroyed with each call), which can affect performance.

A RESOURCE_LOCAL EntityManager provides its clients with an EntityTransaction object for managing transactions. The commitTransaction() method in Listing demonstrates its use, and the implicit

behavior of this facade is achieved by the policy of calling commitTransaction() at the end of every method that updates the persistence context (through a persist, merge, or remove operation).

Filtering Test Data Using a CMT Session Bean

After using the Java facade (through a utility class) to populate the demo data, the servlet client then calls a stateless CMT session bean, OrderProcessorCMTBean, to filter this data remove Customer and Wine entities that might have been created from previous invocations.

We could have both data population and data filtering with a single facade tied to a single persistence unit, but we are deliberately mixing and matching options here to show how they can interact.

What allows this to work is that both persistence units, one RESOURCE_LOCAL and one JTA, both point to the same database connection.

/ Filter the data by removing any existing Customers with email 'xaction.head@yahoo.com'
// and any existing Wine with country 'United States'. out.print("<h2>Filtering Demo Data... "); System.out.println(orderProcessorCMT.initialize()); out.println("done</h2>");

The stateless session bean OrderProcessorCMTBean does not explicitly declare its transactional behavior, and so it assumes the default TransactionManagement value— CMT—which is the equivalent of annotating the bean:

@TransactionManagement(TransactionManagementType.CONTAINER)

Because the initialize() method is not annotated with a TransactionAttribute override, and OrderProcessorCMTBean does not override the default TransactionAttribute value for all its methods at the bean level, it assumes the default transaction attribute value, the equivalent of the following:

@TransactionAttribute(TransactionAttributeType.REQUIRED)

Since the client has neither begun nor inherited a transaction, one is created and begun by the EJB

container for the duration of the initialize() method and all changes are committed upon successful completion of this method.

CMT beans will always exhibit implicit commit behavior because the container will not allow a transaction that it begins to continue after the method has completed.

The implicit commit causes any changes made during the course of that method to be made persistent, and they are applied to the database so that the changes are visible to all clients henceforth.

Creating New Customer and CartItem Entity Instances in the Client

The next step for the client is to create a new Customer entity instance (actually, the concrete Individual entity subclass of the abstract Customer entity), create some Wine instances, and add some bottles of wine represented by CartItem instances, to the customer's cart:

```
// Create a Customer and add some CartItems and their associated Wines Individual customer = new Individual();
customer.setFirstName("Transaction");
customer.setLastName("Head"); customer.setEmail(PopulateDemoData.TO_EMAIL_ADDRESS);
for (int i = 0; i < 5; i++) { final Wine wine = new Wine(); wine.setCountry("United States");
wine.setDescription("Delicious wine"); wine.setName("Xacti"); wine.setRegion("Dry Creek Valley"); wine.setRetailPrice(new Float(20.00D + i));
 wine.setVarietal("Zinfandel"); wine.setYear(2000 + i); orderProcessorCMT.persistEntity(wine);
final CartItem cartItem = new CartItem();
cartItem.setCreatedDate(new Timestamp(System.currentTimeMillis()));
cartItem.setCustomer(customer);
cartItem.setQuantity(12);
cartItem.setWine(wine);
customer.addCartItem(cartItem);
}
```

Note that during this stage, only the wine instances are persisted explicitly. All other entities that are created are associated, directly or indirectly, with the customer instance, and they exist only in the servlet's method context.

No transaction is involved in this process of creating these entity objects, assigning their ordinary properties, and associating them with each other. The wine instances are deliberately not associated with the other object through cascade rules, so they must be persisted explicitly.

Persisting the Customer

Having created the Customer and associated CartItem objects, the client passes the Customer to the OrderProcessorCMT bean's persistEntity() method. Because the relationships on the Customer and CartItem entities are annotated cascade

```
= {CascadeType.ALL}, the act of persisting the Customer entity is cascaded to all

associated entities, and so they are all persisted as well. This method call will begin a transaction, persist the customer and related objects to the databas
e, and commit the work:

// Persist the Customer, relying on the cascade settings to persist all related CartItem entities as well.
After the call, the Customer instance will have an ID value that was assigned by the EJB container when it was persisted.

orderProcessorCMT.persistEntity(customer);
```

Also note that because we set up an ID generator on the base class for each of the entities in our persistence unit, their id fields are auto-populated at the time they are persisted, and foreign key columns mapped to entity relationships will be wired up properly as well.

For the BusinessContext class, which uses a table generator, the ID generator for its id field is defined like this:

```
@Id
@Column(nullable = false)
@GeneratedValue(strategy = GenerationType.TABLE, generator = "BusinessContact_ID_Generator") private Integer id;
```

The Customer instance we pass to persistEntity() is updated to become both managed and persisted. Were we calling persistEntity() through a remote interface on OrderProcessorCMTBean, the invocation would use pass-by-value semantics, and we would need to capture the updated Customer instance in the method result.

Since we’re calling the session bean using local mode from within the Java EE tier, we are using pass- by-reference semantics, so the Customer instance is updated directly.

At the conclusion of the persistEntity() call, the Customer (Individual) and all associated data is now applied to the database and available to all clients, including our own.

Creating the CustomerOrder

An instance of a Customer entity now exists as a persistent row in the database, so we can call createCustomerOrder() with the customer to create a new CustomerOrder, and **create an OrderItem for each CartItem on the Customer:**

```
// Create a customer order and create OrderItems from the CartItems final CustomerOrder customerOrder =

orderProcessorCMT.createCustomerOrder(customer);
```

Here again, the createCustomerOrder() method definition is not annotated with a transaction attribute, so it defaults to REQUIRED, and the EJB container creates and begins a new transaction for the

duration of that method, and then commits it upon returning control to the client.

Note that the implementation of the createCustomerOrder() method delegates to another method, createCustomerOrderUsingSupports() , which is annotated as follows:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)

public CustomerOrder createCustomerOrderUsingSupports(Customer customer) {...}
```

This delegation exists purely to allow us to illustrate the transaction behavior involved when calling a method marked SUPPORTS from a method marked REQUIRED.

The method called from the client, createCustomerOrder(), causes a transaction to be created that is propagated to its delegate, createCustomerOrderUsingSupports().

This latter method inherits the transaction context created by the EJB container for its caller.

Had the client called createCustomerOrderUsingSupports() directly, an exception would have been thrown during its execution, when the remove() and persist() operations were called outside a transaction context.

A lot is going on inside the createCustomerOrderUsingSupports() method. Because the customer argument might be detached (in our case it isn't, since our servlet is running within a local Java EE environment), it needs to be turned into a managed instance. If it is already managed, this precaution is unnecessary but not harmful:

```
if (!em.contains(customer)) {

customer = em.merge(customer);

}
```

Next, the CustomerOrder instance is created and added to the Customer. Our implementation of the addCustomerOrder() method adds the CustomerOrder to the Customer's customerOrderList property and also assigns the customer back-pointer property on CustomerOrder, effectively wiring up the bidirectional relationship:

```
final CustomerOrder customerOrder = new CustomerOrder(); customer.addCustomerOrder(customerOrder);
```

And the CustomerOrder is then populated with new OrderItems to match each CartItem in the Customer's shopping cart. We copy the customer's CartItem list into an ArrayList so we can iterate over it.

And remove each CartItem from the Customer after a corresponding OrderItem has been created in the CustomerOrder, without causing a concurrency exception:

```
final Timestamp orderDate = new Timestamp(System.currentTimeMillis()); final List<CartItem> cartItemList =
new ArrayList(customer.getCartItemList()); for (CartItem cartItem : cartItemList) {
// Create a new OrderItem for this CartItem final OrderItem orderItem = new OrderItem(); orderItem.setOrderDate(orderDate); orderItem.setPrice(cartItem.
getWine().getRetailPrice());
orderItem.setQuantity(cartItem.getQuantity());
orderItem.setStatus("Order Created");
orderItem.setWine(cartItem.getWine());
customerOrder.addOrderItem(orderItem);
// Remove the CartItem customer.removeCartItem(cartItem);
}
```

As each OrderItem is created, its CartItem is removed from the Customer. An orphanRemoval=true property on the @OneToMany relationship annotating Customer. orderItemList ensures that after each CartItem is removed from the Customer, it will be automatically removed from persistent storage as well when the context transaction is committed.

At last, the newly populated CustomerOrder is persisted and returned to the caller:

```
return persistEntity(customerOrder);
```

The transaction is not committed until after the createCustomerOrderUsingSupports() method has completed and control is returned from the wrapper createCustomerOrder() method.

Assuming that we are using a JTA transaction that uses a two-phase commit (for instance, a container-managed transaction created by the EJB container), should anything go wrong in the course of either of these methods, the entire transaction will be rolled back, and neither this client nor any outside application will ever be aware that a CustomerOrder was created.

Does This Pass the Acid Test?

Have the core ACID requirements that characterize a valid transaction been met? Let's look at how EJB addresses each one.

Atomicity

The EJB container ensures that whenever a stateless CMT method marked REQUIRED or REQUIRES_NEW is called, if the container interposes to create a new transaction (this will always happen with REQUIRES_NEW), it will resolve that transaction upon exiting the method.

If the method completes successfully, and if the bean code did not call EJBContext.setRollbackOnly(), the transaction will be committed. If the method throws an exception, or if EJBContext.setRollbackOnly() is called, the transaction will be rolled back.

These two transaction attributes are the only ones for which the container may interpose to create a new transaction.

For all other transaction attributes, either an externally managed transaction is involved (in which case the container will not interpose to commit it when the method is exited), or the method is called with no transaction context.

In the latter case, because we are using a transaction-scoped persistence context and there is no transactional context, calls to persist(), merge(), or remove() will cause a javax.persistence.TransactionRequiredException.

Were we to be using a stateful session bean with an extended persistence context, these changes would be tracked by the persistence context, even outside of a transaction, and applied should a transaction be subsequently created (and associated with this persistence context) and committed.

Consistency

Any database constraints or concurrency conditions (whether enforced in the database or in the EJB container) are guaranteed to be satisfied when a transaction is committed through the EJB services.

Violations will result in exceptions being thrown from the EJB container, and the transaction will automatically be rolled back. A successful commit indicates that all defined constraint conditions have been met.

Isolation

This requirement is largely the responsibility of the underlying JTA resources. Each resource may expose its own configurable isolation level settings to provide varying degrees of consistency to the resources involved in a transaction.

Isolation levels determine the extent to which resources within the transaction are able to see the partial (in-transaction) state of other resources involved in the transaction and largely translate into cache consistency settings within the resource.

Isolation also determines that the transaction should not see uncommitted data of another transaction. To remain database neutral, our example did not attempt to configure these settings.

Durability

This is also largely the responsibility of the underlying JTA resources involved in the transaction (e.g., the database or mail server). At the conclusion of a JTA transaction, any such resources are expected to be able to show the new state of the data when queried.

We demonstrated this by querying the details of the new CustomerOrder from the client after the createCustomerOrder() method, and its transaction encapsulated within had completed.

Benefits of This Approach

A principal benefit of using a default stateless session bean with CMT demarcation is that the client does not need to be concerned about the beginning, ending, or otherwise coordinating the transaction logic.

Also, any transaction context currently in effect on the thread in which the bean method is called is automatically propagated to that method call (if the transaction attribute is REQUIRED or SUPPORTS).

Each call it makes to the OrderProcessorCMT bean either completes successfully (in which case it can be assumed that the work has been applied persistently) or results in an exception (whereupon the work performed in that method is completely rolled back). It's a very simple model.

Limitations of This Approach

In general, simple is good, but sometimes it is too limiting. While this example allows the client to create and manipulate new entity instances in the client tier, as when it created the Customer, Wine and CartItem instances.

The client must rely on the EJBs to ensure that the updates it makes to the entity model are persisted to the database within a transaction since transactions are always begun and terminated by the EJB container before control is handed back to the client.

In the next example, we will show how using stateful session beans, coupled with BMT and an extended persistence context, empowers the client with greater flexibility (and with it, responsibility) over the transactional behavior of the application.

Note There has been a popular conception among EJB users that stateful session beans should be avoided for performance reasons. The performance tests that we have done strongly suggest that stateful session beans have been falsely maligned and that when correctly used, they can actually boost performance.

Furthermore, in EJB, their value is increased, since they provide you this PersistenceContext.EXTENDED option, allowing entity instances to be cached for use across transactions.

Stateful Session Beans with BMT Demarcation and Extended Persistence Context

To illustrate extending the reach of EJB's transaction support, here is the same application written using a stateful BMT session bean, OrderProcessorBMTBean.java.

This BMT example also leverages EJB's built-in transaction support, but instead of relying on the container to manage transaction demarcation at method boundaries, shows how to demarcate transactions explicitly, inside the enterprise bean code and controlled from the client as well.

There is no requirement that you use BMT demarcation when using stateful session beans, and in fact, this option is not typically used.

Listing OrderProcessorBMTBean.java, a Stateful Session Bean Using BMT Demarcation and an Extended Persistence Context

```
@Stateful(name = "OrderProcessorBMT", mappedName = "Blog08- TransactionSamples-OrderProcessorBMT") @TransactionManagement(Transaction ManagementType.BEAN) @Interceptors(OrderProcessorBMTBeanTxnInterceptor.class)
```

```
public class OrderProcessorBMTBean {
@Resource
SessionContext sessionContext;
@PersistenceContext(unitName = "Blog08-TransactionSamples-JTA", type = PersistenceContextType.EXTENDED) private EntityManager em;
/**
* Remove any existing Customers with email 'wineapp@yahoo.com' and any
* existing Wine with country 'United States'
*/
public String initialize() throws HeuristicMixedException, HeuristicRollbackException,
RollbackException,
SystemException {
StringBuffer strBuf = new StringBuffer(); strBuf.append("Removed "); int i = 0;
// Filter the data by removing any existing Customers with email
// 'wineapp@yahoo.com' (or whatever is defined in the user.properties file).
// The first call to a transactional method on OrderProcessorBMT will begin a
// transaction.
for (Customer customer
: getCustomerFindByEmail(PopulateDemoData.TO_EMAIL_ADDRESS)) { em.remove(customer);
i++;
}
strBuf.append(i);
strBuf.append(" Customer(s) and ");
// Remove any existing Wine with country 'United States' i = 0;
for (Wine wine : getWineFindByCountry("United States")) { em.remove(wine);
i++;
}
strBuf.append(i);
strBuf.append(" Wine(s)");
// Apply these changes, committing the entity removal operations commitTransaction();
return strBuf.toString();
}
/**
* Create a new CustomerOrder from the items in a Customer's cart. Creates a new CustomerOrder entity, and then creates a new OrderItem entity for ea
ch CartItem found in the Customer's cart.
* Using CMT w/ the default Required xaction attribute, if this method is invoked without a transaction context, a new transaction will be created by the EJB
container upon invoking the method, and committed upon successfullycompleting the method.
*/
public CustomerOrder createCustomerOrder(Customer customer) throws Exception { if (customer == null) {
throw new IllegalArgumentException("OrderProcessingBean.
createCustomerOrder(): Customer not specified");
}
// Ensure we are working with a managed Customer object customer = em.find(Customer.class, customer.getId());
CustomerOrder customerOrder = new CustomerOrder(); customer.addCustomerOrder(customerOrder);
final Timestamp orderDate = new Timestamp(System.currentTimeMillis());
// Clone the CartItem list so we remove the CartItem entries from the Customer
// without causing a ConcurrentModificationException on the iterator.
final List<CartItem> cartItemList = new ArrayList(customer.getCartItemList()); for (CartItem cartItem : cartItemList) {
// Create a new OrderItem for this CartItem final OrderItem orderItem = new OrderItem(); orderItem.setOrderDate(orderDate); orderItem.setPrice(cartItem.
getWine().getRetailPrice()); orderItem.setQuantity(cartItem.getQuantity()); orderItem.setStatus("Order Created"); orderItem.setWine(cartItem.getWine()); c
ustomerOrder.addOrderItem(orderItem);
// Remove the CartItem. Note that the 'orphanRemoval' flag will ensure
// that the cartItem is removed from the database once it is disassociated
// from a customer.
customer.removeCartItem(cartItem);
}
// The Cascade rules on Customer will cause the CustomerOrder to be
// persisted when the Customer is merged
em.merge(customer);
return customerOrder;
}
@ExcludeClassInterceptors
public void commitTransaction() throws HeuristicMixedException, HeuristicRollbackException, RollbackException, SystemException {
final UserTransaction txn = sessionContext.getUserTransaction(); if (txn.getStatus() == Status.STATUS_ACTIVE) {
txn.commit();
}
}
@ExcludeClassInterceptors
public void rollbackTransaction() throws SystemException {
final UserTransaction txn = sessionContext.getUserTransaction(); if (txn.getStatus() == Status.STATUS_ACTIVE) {
txn.rollback();
}
}
@ExcludeClassInterceptors
public boolean isTransactionDirty() throws SystemException { final UserTransaction txn = sessionContext.getUserTransaction(); return Boolean.valueOf(t
xn.getStatus() == Status.STATUS_ACTIVE);
}
```

```
}
@ExcludeClassInterceptors
public Object queryByRange(String jpqlStmt, int firstResult, int maxResults) {
Query query = em.createQuery(jpqlStmt); if (firstResult > 0) {
query = query.setFirstResult(firstResult);
}
if (maxResults > 0) {
query = query.setMaxResults(maxResults);
}
return query.getResultList();
}
public <T> T persistEntity(T entity) { em.persist(entity);
return entity;
}
public <T> T mergeEntity(T entity) {
return em.merge(entity);
}
public <T> void removeEntity(T entity) { em.remove(em.merge(entity));
}
@ExcludeClassInterceptors
public <T> List<T> findAll(Class<T> entityClass) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
413
Blog 8 Transaction Management
cq.select(cq.from(entityClass));
return em.createQuery(cq).getResultList();
}
@ExcludeClassInterceptors
public <T> List<T> findAllByRange(Class<T> entityClass, int[] range) { CriteriaQuery cq = em.getCriteriaBuilder().createQuery(); cq.select(cq.from(entityC
lass));
Query q = em.createQuery(cq);
q.setMaxResults(range[1] - range[0]);
q.setFirstResult(range[0]);
return q.getResultList();
}
/**
* <code>select o from Customer o where o.email = :email</code>
*/
@ExcludeClassInterceptors
public List<Customer> getCustomerFindByEmail(String email) {
return em.createNamedQuery("Customer.findByEmail", Customer.class). setParameter("email", email).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.year = :year</code>
*/
@ExcludeClassInterceptors
public List<Wine> getWineFindByYear(Integer year) {
return em.createNamedQuery("Wine.findByYear", Wine.class).
setParameter("year", year).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.country = :country </code>
*/
@ExcludeClassInterceptors
public List<Wine> getWineFindByCountry(String country) { return em.createNamedQuery("Wine.findByCountry", Wine.class). setParameter("country", co
untry).getResultList();
}
/**
* <code>select object(wine) from Wine wine where wine.varietal = :varietal</code>
*/
@ExcludeClassInterceptors
public List<Wine> getWineFindByVarietal(String varietal) { return em.createNamedQuery("Wine.findByVarietal", Wine.class). setParameter("varietal", vari
etal).getResultList();
}
/**
* <code>select o from InventoryItem o where o.wine = :wine</code>
*/
@ExcludeClassInterceptors
public List<InventoryItem> getInventoryItemFindItemByWine(Object wine) { return em.createNamedQuery("InventoryItem.findItemByWine", InventoryItem
.class).setParameter("wine", wine).getResultList();
}
}
```

Coupled with this stateful session bean is an Interceptor class that serves to interpose on each method that applies changes through the EntityManager to automatically begin a transaction.

This pattern is similar to SQL’s transactional model, where a transaction is implicitly begun each time a DML operation like INSERT, UPDATE, or DELETE is called. Hence, the client is responsible only for calling COMMIT or ROLLBACK to conclude the transaction, but there is no explicit call to begin the transaction.

Listing OrderProcessorBMTBeanTxnInterceptor.java, an Interceptor used by OrderProcessorBMTBean to begin a JTA transaction, using the UserTransaction from the BMT bean’s SessionContext, each time a method is called which applies changes through the EntityManager

```
class OrderProcessorBMTBeanTxnInterceptor { public OrderProcessorBMTBeanTxnInterceptor() {
}
@AroundInvoke
Object beginTrans(InvocationContext invocationContext) throws Exception { final OrderProcessorBMTBean orderProcessorBMTBean = (OrderProcessor
BMTBean) invocationContext.getTarget();
final UserTransaction txn = orderProcessorBMTBean.sessionContext. getUserTransaction();
if (txn.getStatus() == Status.STATUS_NO_TRANSACTION) { txn.begin();
}
return invocationContext.proceed();
}
}
```

Listing shows OrderProcessorBMTClient.java, a servlet client that drives the OrderProcessorBMT session bean to demonstrate EJB’s BMT demarcation by calling through the UserTransaction interface.

Listing OrderProcessorBMTClient.java, Our Mock Java SE Client

```
@WebServlet(name = "OrderProcessorBMTClient", urlPatterns = {"/ OrderProcessorBMTClient"})
public class OrderProcessorBMTClient extends HttpServlet { @EJB
OrderProcessorBMTBean orderProcessorBMT;
/**
* Processes requests for both HTTP
* <code>GET</code> and
* <code>POST</code> methods.
*
* @param request servlet request
* @param response servlet response
* @throws ServletException if a servlet-specific error occurs
* @throws IOException if an I/O error occurs
*/
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException { response.setContentType("text/html;charset=UTF-8"); response.setContentType("text/html;charset=UTF-8"); Out
putStream rOut = response.getOutputStream(); PrintStream out = new PrintStream(rOut);
try {
/* TODO output your page here. You may use following sample code. */
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet OrderProcessorBMTClient</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Servlet OrderProcessorBMTClient at " + request.
getContextPath() + "</h1>");
out.println("</body>");
out.println("</html>");
out.print("<h2>Populating Demo Data... ");
PopulateDemoData.resetData("Blog07-WineAppUnit-ResourceLocal",
System.out);
out.println("done</h2>");
out.print("<h2>Filtering Demo Data... "); StringBuffer strBuf = new StringBuffer(); strBuf.append("Removed "); int n = 0;
```

```
// Filter the data by removing any existing Customers with email
// 'wineapp@yahoo.com' (or whatever is defined in the user. properties file).
// The first call to a transactional method on OrderProcessorBMT will begin a
// transaction.
for (Customer customer :
orderProcessorBMT.getCustomerFindByEmail(PopulateDemoData.TO_
EMAIL_ADDRESS)) {
orderProcessorBMT.removeEntity(customer);
n++;
}
strBuf.append(n + " Customer(s) and ");
// Remove any existing Wine with country 'United States' n = 0;
for (Wine wine : orderProcessorBMT.getWineFindByCountry("United States")) {
orderProcessorBMT.removeEntity(wine);
n++;
}
strBuf.append(n + " Wine(s)"); out.print(strBuf.toString() + "</h2>");
// Apply these changes, committing the entity removal operations orderProcessorBMT.commitTransaction();
// Create a Customer and add some CartItems and their associated Wines Individual customer = new Individual(); customer.setFirstName("Transaction");
customer.setLastName("Head"); customer.setEmail(PopulateDemoData.TO_EMAIL_ADDRESS);
for (int i = 0; i < 5; i++) { final Wine wine = new Wine(); wine.setCountry("United States"); wine.setDescription("Delicious wine"); wine.setName("Xacti");
wine.setRegion("Dry Creek Valley"); wine.setRetailPrice(new Float(20.00D + i)); wine.setVarietal("Zinfandel"); wine.setYear(2000 + i); orderProcessorBM
T.persistEntity(wine);
final CartItem cartItem = new CartItem();
cartItem.setCreatedDate(new Timestamp(System.currentTimeMillis()));
cartItem.setCustomer(customer);
cartItem.setQuantity(12);
cartItem.setWine(wine);
customer.addCartItem(cartItem);
}
// Persist the Customer, relying on the cascade settings to persist all
// related CartItem entities as well. After the call, the Customer
// instance will have an ID value that was assigned by the EJB container
// when it was persisted.
orderProcessorBMT.persistEntity(customer);
// Create a customer order and create OrderItems from the CartItems final CustomerOrder customerOrder =
orderProcessorBMT.createCustomerOrder(customer);
out.print("<h2>Retrieving Customer Order Items... ");
for (OrderItem orderItem : customerOrder.getOrderItemList()) { final Wine wine = orderItem.getWine(); out.println(wine.getName() + " with ID " + wine.getI
d());
}
out.println("done</h2>");
// Commit the order, applying all of the changes made thus far orderProcessorBMT.commitTransaction();
} catch (Exception ex) { ex.printStackTrace();
if (orderProcessorBMT != null) { try {
orderProcessorBMT.rollbackTransaction();
} catch (Exception e) { e.printStackTrace();
}
}
} finally { rOut.close(); out.close();
}
}
/* HTTPServlet methods... */
}
```

Transaction Analysis

The following sections will analyze this second example from a transactional perspective. We have empowered the session bean with the state (i.e., Stateful), giving it control over the demarcation of its transactions, and allowed its associated persistence context to survive from one transaction to the next.

Session Bean Declaration

These features have arisen through the combination of annotations and code. You'll notice that this session bean is annotated:

```
@Stateful(name = "OrderProcessorBMT", mappedName = "Blog08- TransactionSamples-OrderProcessorBMT") @TransactionManagement(Transaction
ManagementType.BEAN) @Interceptors(OrderProcessorBMTBeanTxnInterceptor.class) public class OrderProcessorBMTBean {
@Resource
SessionContext sessionContext;
@PersistenceContext(unitName = "Blog08-TransactionSamples-JTA", type = PersistenceContextType.EXTENDED) private EntityManager em;
...
}
```

It injects both a SessionContext and an EntityManager. Being stateful allows the enterprise bean to retain state from one client invocation to the next. In this case, that state is the persistence context and the associated transaction, which must survive through multiple method invocations.

The BMT declaration means that the container should not automatically interpose on method boundaries to demarcate transactions. Attempts to add TransactionAttribute qualifiers to methods on a BMT session bean will be caught and raise an exception at deployment time.

The @PersistenceContext annotation holds a type property with value PersistenceContextType.EXTENDED, meaning that it persists from one transaction to the next, and

allows associated entities to remain managed even after the transaction in which they were created has ended.

The UserTransaction object available through the sessionContext property is this BMT bean's interface onto the EJB container's JTA transaction manager and exposes the begin(), commit(), and rollback() transaction demarcation methods.

Removing Previous Test Data

We could have populated the test environment through a session bean method call, as we did for the preceding CMT example. However, using BMT offers us the option of performing this work interactively, in the client.

This is because the OrderProcessorBMT bean's persistence context is EXTENDED, allowing the entities to remain associated with a persistence context even after control has been returned from the enterprise bean to the client.

```
// Filter the data by removing any existing Customers with email
// 'wineapp@yahoo.com' (or whatever is defined in the user.properties file).
// The first call to a transactional method on OrderProcessorBMT will begin a
// transaction.
for (Customer customer :
```

```
orderProcessorBMT.getCustomerFindByEmail(PopulateDemoData.TO_EMAIL_ADDRESS)) { orderProcessorBMT.removeEntity(customer); n++;
}
strBuf.append(n + " Customer(s) and ");
// Remove any existing Wine with country 'United States' n = 0;
for (Wine wine : orderProcessorBMT.getWineFindByCountry("United States")) { orderProcessorBMT.removeEntity(wine);
n++;
}
strBuf.append(n + " Wine(s)"); out.print(strBuf.toString() + "</h2>");
// Apply these changes, committing the entity removal operations orderProcessorBMT.commitTransaction();
```

Each call to removeEntity() is performed in the transaction that was begun on the OrderProcessorBMT bean through its Interceptor and puts the entity in the “removed” state in its persistence context.

At the conclusion of these steps, the client calls commitTransaction() to actually perform the DBMS DELETE operations in the database and commit the transaction.

Creating New Customer and CartItem Entity Instances in the Client

As with the preceding stateless session example, the step of instantiating the Customer and its CartItem entity instances and wiring them all together involves no transactions, and they can be carried out entirely within the client:

```
// Create a Customer and add some CartItems and their associated Wines Individual customer = new Individual(); customer.setFirstName("Transaction");
customer.setLastName("Head"); customer.setEmail(PopulateDemoData.TO_EMAIL_ADDRESS);
for (int i = 0; i < 5; i++) { final Wine wine = new Wine(); wine.setCountry("United States"); wine.setDescription("Delicious wine"); wine.setName("Xacti"); win
e.setRegion("Dry Creek Valley"); wine.setRetailPrice(new Float(20.00D + i));
wine.setVarietal("Zinfandel"); wine.setYear(2000 + i); orderProcessorBMT.persistEntity(wine);
final CartItem cartItem = new CartItem();
cartItem.setCreatedDate(new Timestamp(System.currentTimeMillis()));
cartItem.setCustomer(customer);
cartItem.setQuantity(12);
cartItem.setWine(wine);
customer.addCartItem(cartItem);
}
```

It is worth noting that prior to JPA, this work would have required much more effort and expended more resources. With EJB 2.x entity beans, the client developer had two main options.

Under one approach, the developer could create data transfer objects (DTOs) or follow some other similar pattern to simulate the task of creating and associating the entity objects through proxies.

This network of DTO classes would then be passed into the session bean layer, as we did previously; but inside the session bean, actual entity beans would have to be explicitly created and initialized from the DTO objects.

A second approach, updating the entity beans directly from the client, is simpler to code, but potentially at the expense of higher performance costs.

If the client exists outside the Java EE tier, each method call would incur the overhead of RMI/IIOP (remote method invocation over the Internet inter-ORB protocol) marshaling to communicate with the

actual EJB object residing in the EJB container.

Much of this overhead is removed when the client lives in the Java EE tier since it could use local entity bean interfaces to communicate directly with the live entity bean, but Java SE clients were forced to use remote interfaces onto the entity beans.

On top of that, container-managed relationships (CMR) are only supported on local component interfaces; so direct entity bean relationship lookups and updates were not even available to Java SE clients in the EJB 2.x world.

Persisting the Customer

Although we chose in this example to embed the transaction begin() operation inside the Interceptor class that interposes on the BMT session bean's methods, we could have exposed a beginTransaction() call to the client as well.

Because we have chosen the approach we did, all that is required is the call to persistEntity() that now implicitly begins the transaction (but does not commit it):

// Persist the Customer, relying on the cascade settings to persist all related CartItem entities as well. Reassign the customer to pick up the ID value that was assigned by the EJB container when it was persisted.

```
orderProcessorBMT.persistEntity(customer);
```

The transaction context does not extend to the client thread itself; it exists only in the session bean's thread. The call to UserTransaction.begin() that occurs inside the Interceptor establishes a transaction context on that thread that is then available to the session bean when its persistEntity() method is called.

Creating the CustomerOrder

Our transaction, now in effect, continues through the step of creating the customer order. This stage is similar to the stateless CMT example except that the transaction has already been created and must be explicitly committed at the conclusion.

```
// Create a customer order and create OrderItems from the CartItems final CustomerOrder customerOrder =
orderProcessorBMT.createCustomerOrder(customer);
out.print("<h2>Retrieving Customer Order Items... ");
for (OrderItem orderItem : customerOrder.getOrderItemList()) { final Wine wine = orderItem.getWine(); out.println(wine.getName() + " with ID " + wine.getI
d());
}
out.println("done</h2>");
// Commit the order, applying all of the changes made thus far orderProcessorBMT.commitTransaction();
```

Should the client wish to cancel the order at this stage, perhaps through interactive confirm/cancel buttons exposed in a client panel, the BMT option provides this possibility even after the CustomerOrder has been created.

Benefits of This Approach

The benefit of using explicit transaction demarcation is the additional degree of flexibility that it offers. The EJB server is still acting in its capacity as a transaction manager, only it exposes the transaction demarcation control to the enterprise bean instead of automating this demarcation based on the @TransactionAttribute settings on each method.

While the stateless example could have prompted the user before creating the CustomerOrder, this approach allows the CustomerOrder to be created and validated— for example, before being submitted to the user for confirmation. BMT must be used with caution, however, for the reasons mentioned in the following section.

Limitations of This Approach

It can be argued that the additional degree of flexibility is typically outweighed by the additional burdens of tracking the transaction state and avoiding misuse by session bean clients.

Leaving the process of beginning and ending transactions to the mercy of the order in which clients call the session bean methods offers the possibility of dangling transactions.

The client, in coordination with the bean itself, has the responsibility of cleanly ending—whether committing or rolling back—each transaction that has begun.

This may be a reasonable risk if you can control how clients will use the bean—but session beans are openly published, and it may be difficult to anticipate who might use them, and how.

BMT session beans can be written to safeguard against misuse, but this safeguard code is probably going to leave the bean with behavior similar to CMT beans anyway, in which case little is gained for your efforts.

The building, Deploying, and Testing: A Transactional Scenario from the Wines Online Application. Now that we've examined many of the details of transaction support offered by EJB, let's execute the test cases we just covered.

For both the CMT and BMT scenarios, we invoke a servlet client that creates a new customer, builds up a shopping cart consisting of cart item entries, and then creates a customer order consisting of order items based on the cart items in the cart.

We chose these examples for this notes because they involve multiple operations that can be partitioned into transactional work units of greater or less granularity depending on the requirements of the client.

To illustrate the default support provided by EJB, we use the example we explored first: a standard stateless session bean implementation that uses the default CMT demarcation.