Auction Persistence 1.0 Component Specification

1. Design

The Auction Persistence component provides the Orpheus application with an interface to persistent storage of auction data. The central persistence functionality is handled by a stateless session EJB, but for interoperation with the Auction Framework component the bean is wrapped in an ordinary class. The expected usage is for only the client to be used. The EJBs would not be accessed directly.

1.1 Design considerations

There are no major considerations. This is a basically straightforward implementation of an EJB-enabled persistence application.

One topic needs to be discussed, and it is the issue regarding the serializability of value objects. Neither the Auction nor the Bid interfaces are serializable. As such, we need to introduce parallel classes that are serializable. Now, we do control CustomBid, and we very well could simply make it serializable and use it as the Bid data transfer object, but this component is flexible enough to allow future Bid instances to be used instead, as part of an upgrade to the model, and it might not be serializable. Assuredly, the AuctionTranslator would change as well as the AuctionDAO, but the business layer would remain intact. By using our own DTOs, we provide complete reliability of the transfer layer as far as serializability is concerned.

1.2 Design Patterns

1.2.1 Data Transfer Object J2EE Pattern

The AuctionDTO and BidDTO can be said to use the **Data Transfer Object** pattern, but their goal is not to improve performance but rather to provide control over serializability of the data as the Auction and Bid interfaces are not serializable.

1.2.2 Strategy

Used extensively in this design. The AuctionDAO and AuctionTranslator interfaces are part of the **Strategy** pattern. Even the Auction and Bid interfaces are handled in this way, although we do provide specific AuctionTranslators for them. Needless to say, but the EJB interfaces also use this pattern.

1.2.3 Factory pattern

The AuctionDAOFactory uses this pattern, to some degree, to provide a single entry point for EJBs to obtain the AuctionDAO instances. In a more orthodox usage of this pattern, the factory would accept a token and retrieve a type of DAO instance based on this token. This factory has a specific method to return DAO of a single type, and they don't manufacture new DAO instances with every call. IN reality, the true factory here is the ObjectFactory class.

1.2.4 DAO

The AuctionDAO class uses the **DAO** pattern to encapsulate data access in one abstracted place.

1.2.5 Template method

The abstract client CustomAuctionPersistence implement the work of caching data and translating between objects, but leave the details of interfacing with the persistence to the sub-classes using **Template Method** pattern.

1.3 Industry Standards

JDBC, SQL Server 2000 T-SQL, EJB 2.1

1.4 Required Algorithms

There are no complex algorithms here. Implementation hints are provided in Poseidon documentation as “Implementation Notes” sections.

For mapping of the data to the database tables, please see section 2.1.2.2 and 2.1.2.3 of the Requirements Specification.

1.5 Component Class Overview

The component classes and interfaces are spread across three packages. There is the main package where the clients and data access classes are located. The EJBs are located in a separate sub-package, as are the specific data access and translator classes.

1.5.1 com.orpheus.auction.persistence

This is the main package. It contains the clients and data access classes are located

CustomAuctionPersistence
This is the auction persistence client to the EJB layer. It implements the AuctionPersistence and supports all operations. It maintains a cache for faster performance. It uses the ConfigManager and Object Factory to initialize itself. It is built to work with EJBs, and this class leaves it to implementations to specify the EJBs. Hence the abstract ejbXXX methods. The public methods defer to these for actual persistence calls. It delegates to AuctionTranslator instance to perform translations between the Auction Framework’s Auction and Bid instances, and their equivalent transport entities – AuctionDTO and BidDTO.

LocalCustomAuctionPersistence
Implements the abstract ejbXXX methods to work with the local auction EJB. Simply defers all calls to the EJB. It uses the ConfigManager and Object Factory to initialize the JNDI EJB reference to obtain the handle to the EJB interface itself.

RemoteCustomAuctionPersistence
Implements the abstract ejbXXX methods to work with the remote auction EJB. Simply defers all calls to the EJB. It uses the ConfigManager and Object Factory

to initialize the JNDI EJB reference to obtain the handle to the EJB interface itself.

AuctionDAOFactory
Static factory for supplying the auction DAO instance to the EJBs. It uses synchronized lazy instantiation to get the initial instance of each DAO. Supports the creation of the AuctionDAO.

AuctionDAO
Interface specifying the methods for Auction persistence. Works with the DTO version of the Auction. Supports all methods in the client.

AuctionTranslator
Interface specifying the contract for translating between Auction and Bid instances and their transport equivalents – AuctionDTO and BidDTO. The former are value objects used on the outside world and a DTO is an entity this component uses to ferry info between the clients and the DAOs. Implementations will constrain the data types they support.

CustomBid
An implementation of the Bid interface to support the addition of a mutable id and immutable imageId. Also, the effectiveAmount is given a setter. This is the Bid implementation that is excepted to be used by the persistence clients.

1.5.2 com.orpheus.auction.persistence.ejb

This is the package where all EJBs and DTOs are located.

AuctionLocalHome
This is the local home interface for managing auctions. The local client will obtain it to get the local interface.

AuctionLocal
This is the local interface used to talk to the AuctionBean. Supports all client operations.

AuctionRemoteHome
This is the remote home interface for managing auctions. The remote client will obtain it to get the remote interface.

AuctionRemote
This is the remote interface used to talk to the AuctionBean. Supports all client operations.

AuctionBean

The EJB that handles the actual client requests. It accepts all client operations, but simply delegates all operations to the AuctionDAO it obtains from the AuctionDAOFactory.

AuctionDTO
Simple transfer bean that exists parallel to the Auction interface but is serializable. It transports the data between the client and the DAO layers. It is assembled at both those ends, and is also cached in the DAO layer. The EJB layer does not operate on it.

BidDTO
Simple transfer bean that exists parallel to the Bid interface but is serializable. Specifically, it works in parallel with the CustomBid class. It transports the data between the client and the DAO layers. It is assembled at both those ends, and is also cached in the DAO layer. The EJB layer does not operate on it.

1.5.3 com.orpheus.auction.persistence.impl

This is the package where the translator and data access implementations are located.

DefaultAuctionTranslator
Implements AuctionTranslator. It translates between the Auction and the AuctionDTO as well as between CustomBid and BidDTO. It is plugged into the CustomAuctionPersistence class to perform these translations. The translation is a simple 1-1 mapping between these entities.

SQLServerAuctionDAO
Implements AuctionDAO. Works with SQL Server database and the tables of auction, bid, and effective_bid, with additional usage of the image, hosting_block, domain, and sponsor tables. The mappings are mostly 1-1, with Auction info going into the auction table, and Bid info goes into the bid and effective_bid tables, still one record in each per Bid. Well, at most one, since a Bid might not have an effective bid. It supports all defined CRUD operations. It also supports caching auctions to minimize SQL traffic. It uses ConfigManager and Object Factory to configure the connection factory and cache instances. It is expected that the former will use a JNDI connection provider so the Datasource is obtained from the application server. It creates, caches, and consumes AuctionDTO objects. Note that AuctionDTO will contain the BidDTOs.

1.6 Component Exception Definitions

This component defines six custom exceptions.

ObjectInstantiationException
This exception is thrown by the constructors of most custom classes in this design that require configuration. The classes that use this exception include:

CustomAuctionPersistence, LocalCustomAuctionPersistence, RemoteCustomAuctionPersistence, and SQLServerAuctionDAO. It is thrown if there is an error during the construction of these objects.

PersistenceException
Extends AuctionException. This exception is the base exception for persistence operations in this component. As such, it and its three subclasses are thrown by the ejbXXX method in the clients, the business methods in the EJBs, and the DAOs. In effect, the client helper method and EJB business methods act as a pass-though for these exceptions.

DuplicateEntryException
Extends PersistenceException. This is a specific persistence exception when inserting a record with a primary id that already exists. The DAO and the associated EJB and client helper methods use it.

EntryNotFoundException
Extends PersistenceException. This is a specific persistence exception when retrieving, updating, or deleting a record with a primary id that does not exist. The DAO and the associated EJB and client helper methods use it.

InvalidEntryException
Extends PersistenceException. This is a specific persistence exception when inserting a record with a primary id that is not valid, which excludes issues such as duplicate ids. The DAO and the associated EJB and client helper methods use it.

TranslationException
Extends AuctionException. This exception is thrown by AuctionTranslator and its subclasses if there is an error while doing the translations. At this point, the DefaultAuctionTranslator does not use it because the translations are very simple and do not involve any checked exceptions.

1.7 Thread Safety

Thread safety is an integral part of this component. But this does not mean all classes are thread-safe. In fact, the DTOs are not, and neither is CustomBid, but more on that later.

The EJBs are not thread safe, but the container assumes responsibility for this, and since these are stateless beans, they hold no state for us, so the status of their thread-safety is not a concern of ours. Transaction control, a related topic, is managed by the container.

All caches, DAOs, clients, and translators are thread-safe inasmuch as they hold no mutable state. The AuctionDAOFactory synchronizes its methods.

The only issue at hand is the lack of thread-safety of the bean and DTO classes mentioned above. This, however, will not affect the component adversely because it is not expected that more than one thread will mutate a CustomBid at a time, and the DTOs are used exclusively by this component, and are not mutated after construction by it.

As such, we can state that in the confines of expected usage of this component, the component maintains effective thread-safety.

To achieve full thread-safety, one would have to make the DTOs and CustomBid classes thread-safe.

2. Environment Requirements

2.1 Environment

JDK 1.4, J2EE 1.4

2.2 TopCoder Software Components

- Configuration Manager 2.1.4
    - Used for configuration in constructors throughout this component to instantiate required classes.
- Object Factory 2.0
    - Used to instantiate classes in constructors throughout this component to instantiate required classes.
- Simple Cache 2.0
    - Used by clients and DAOs to cache auctions for efficiency.
- Auction Framework 1.0
    - Provides the AuctionPersistence interface as well as the Auction and Bid interfaces.
- DBConnection Factory 1.0
    - Provides a convenient access to Connections from a Datasource obtained from JNDI. The implementing DAO classes thus don't have to code their own access to this container resource.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

There are no third party components that need to be used directly by this component.

3. Installation and Configuration

3.1 Package Names

com.orpheus.auction.persistence
com.orpheus.auction.persistence.ejb
com.orpheus.auction.persistence.impl

3.2 Configuration Parameters

3.2.1 All client classes

Parameter	Description	Details
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: "com.topcoder.specify"
translatorKey	Key for the AuctionTranslator to pass to ObjectFactory. Required	Valid key
cacheKey	Key for the Cache to pass to ObjectFactory. Required	Valid key
jndiEjbReference	The JNDI reference for the EJB. Required	Example: "java:comp/env/ejb/AuctionLocal"

3.2.2 AuctionDAOFactory

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: "com.topcoder.specify"
auctionDAO	Key for the AuctionDAO to pass to ObjectFactory. Required	Valid key

3.2.3 SQLServerAuctionDAO

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
factoryKey	Key for the DB Connection Factory to pass to ObjectFactory. Required.	Valid key
name	Name of the connection to the persistence to get from the DB Connection Factory. Optional.	“myConnection” Will use the factory’s default connection, if available, if name not given.
cacheKey	Key for the Cache to pass to ObjectFactory. Required	Valid key

3.3 Dependencies Configuration

3.3.1 DBConnectionFactory, Simple Cache, ConfigManager, ObjectFactory

The developer should refer to the component specification of these components to configure them. The later two are used extensively.

3.3.2 DDL for tables

Please see section 2.1.2.2 of the Requirements Specification.

3.3.3 EJB deployment descriptor

This is provided in the ejb-jar.xml file in the /docs/mappings directory.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Both client and EJBs must be deployed in a container that supports EJB 2.1 standard. The client can be deployed in a thinner environment as long as it has access to the necessary J2EE packages and has access to the Home and Remote interfaces and implementations in the classpath. If using local access, it must be collocated with the EJBs. Overall, this just corresponds to standard EJB usage.

The tables defined in the Requirements Specification must exist, and the database must be running. The ObjectFactory and ConfigManager components must be configured properly.

4.3 Demo

See 4.2 above on the required steps to set up the environment. The demo will only focus on the true public API, which is the clients. The EJBs will not be shown because they are not expected to be used directly. Furthermore, their usage would be very similar to the clients, as the methods are almost completely parallel.

4.3.1 Typical Auction client usage

This demo will demonstrate typical usage of the CustomAuctionPersistence class. This demo will focus on the remote implementation, with the knowledge that usage of the local implementation is the same. Typically this is done in the manager, but we will show method calls directly.

```
// create remote instance with a namespace
CustomAuctionPersistence client = new RemoteCustomAuctionPersistence
(“myNamespace”);

// we might begin by inserting some new auction
Auction auction1 = some auction with valid id: 1
Auction auction2 = another auction with valid id: 2
client.createAuction(auction1);
client.createAuction(auction2);
// at this point, both are inserted and cached at both levels

// we retrieve a cached and non-cached auction, respectively
Auction cAuction1 = client.getAuction(1);
Auction cAuction3 = client.getAuction(3);
// #3 comes from DB since not in cache. After this retrieval it
// is cached

// we now update an auction, after we make some changes (not shown)
client.updateAuction(cAuction3);
// both caches are updated with this

// we now update some bids in auction #2. Some bids will have changes
// (not shown), and others will be new. Yet others will not be part of
// the update but exist in the persistence
Bid[] bids = array of new and some updated Bids for auction #2
client.updateBids(2,bids);
// both caches are updated with the auction #2. It is important to note
// that this updated auction will contain the new bids, updated bids, and
// existing bids that have not been altered in the persistence, and were
// not part of the bids input array.

// we want to delete an auction
client.deleteAuction(1);
// auction deleted from persistence and both caches

// retrieve all auctions between 1/1/2001 and 31/12/2001
Date startDate = date representing 1/1/2001
Date endDate = date representing 31/12/2001
Auction[] auctionsIn2001 = client.findAuctionsByDate(startDate,endDate);
// the client and DAO cache/recache all of these auctions

// retrieve all auctions for a bidder ending no latter than 31/12/2002
long bidderId = valid bidder id: 1
Date endDate2 = date representing 31/12/2002
Auction[] myAuctions = client.findAuctionsByBidder(bidderId,endDate2);
// the client and DAO cache/recache all of these auctions
```

5. Future Enhancements

None at this time.