User Logic Persistence 1.0 Component Specification

1. Design

The User Persistence component provides the Orpheus application with an interface to persistent storage of user data. The central persistence functionality is handled by stateless session EJBs, but for convenient interoperation with a variety of TopCoder components, this is supplemented by an adapter for use of the bean as a persistence plug-in for the User Profile Manager component. The rest of the Orpheus application generally interoperates with this component via the User Profile Manager.

Additionally, this component provides support for pending confirmation storage for the E-mail Validation component. This takes a form similar to the main persistence interface, with a session bean filling the key role behind an adapter that fits it for use with the generic component.

1.1 Design considerations

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

1.2 Design Patterns

1.2.1 Data Transfer Hash J2EE Pattern

The DataTransferObject is a ubiquitous data transfer object that accommodates future changes to the model attributes without altering the bulk of the application transport and client layer.

1.2.2 Strategy

Used extensively in this design. The PendingConfirmationDAO, UserProfileDAO, and ObjectTranslator interfaces are part of the **Strategy** pattern. Needless to say, but the EJB interfaces also use this pattern.

1.2.3 Factory pattern

The DAOFactory uses this pattern, to some degree, to provide a single entry point for EJBs to obtain DAO instances. IN a more orthodox usage of this pattern, the factory would accept a token and retrieve a DAO instance based on this. This factory has specific methods to return DAOs of the right type, and they don't manufacture new DAO instances with every call.

1.2.4 DAO

The PendingConfirmationDAO and UserProfileDAO class use the **DAO** pattern to encapsulate data access in one abstracted place.

1.2.5 Template method

The abstract client OrpheusPendingConfirmationStorage and OrpheusUserProfilePersistence to implement the work of caching data and translating between objects, but leave the details of interfacing with the persistence to the implementations 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.

1.4.1 UserProfile ⇔ UserProfileDTO beans mappings

Table.column	Entity.field
player.user_id	Player.id
player.contact_id	ContactInfo.id
player.payment_pref	Player.paymentPref
admin.user_id	Admin.id
player.user_id	Player.id
sponsor.contact_id	ContactInfo.id
sponsor.fax	Sponsor.fax
sponsor.payment_pref	Sponsor.paymentPref
sponsor.is_approved	Sponsor.approved
user.handle	User.handle
user.e_mail	User.email
user.passwd	User.password
user.is_active	User.active
contact_info.id	ContactInfo.id
contact_info.first_name	ContactInfo.firstName
contact_info.last_name	ContactInfo.lastName
contact_info.address_1	ContactInfo.address1
contact_info.address_2	ContactInfo.address2
contact_info.city	ContactInfo.city
contact_info.state	ContactInfo.state
contact_info.postal_code	ContactInfo.postalCode
contact_info.telephone	ContactInfo.telephone

T
he mappings of the UserProfile to the UserProfileDTO are shown in UserConstants. To access or store the entities in the DTO, please use the following keys:

“player”
“admin”
“sponsor”
“contact_info”

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.user.persistence

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

OrpheusPendingConfirmationStorage
This is the message confirmation client to the EJB layer. It implements the PendingConfirmationStorageInterface and supports all operations. It maintains a cache for faster performance. All exceptions are logged, which is helpful considering that general persistence-related can't be thrown. 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 presence of the abstract ejbXXX methods. The public methods defer to these for actual persistence calls.

LocalOrpheusPendingConfirmationStorage
Implements the abstract ejbXXX methods to work with the local pending confirmation 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.

RemoteOrpheusPendingConfirmationStorage
Implements the abstract ejbXXX methods to work with the remote pending confirmation 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.

OrpheusUserProfilePersistence
This is the user profile client to the EJB layer. It implements the UserProfilePersistence and supports all operations. It maintains a cache for faster performance. All exceptions are logged. 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 presence of the abstract ejbXXX methods. The public methods defer to these for actual persistence calls.

LocalOrpheusUserProfilePersistence
Implements the abstract ejbXXX methods to work with the local user profile 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.

RemoteOrpheusUserProfilePersistence

Implements the abstract ejbXXX methods to work with the remote user profile 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.

DAOFactory
Static factory for supplying the DAO instances to EJBs. It uses synchronized lazy instantiation to get the initial instance of each DAO. Supports the creation of the PendingConfirmationDAO and UserProfileDAO.

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

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

ObjectTranslator
Interface specifying the contract for translating between a value object normally used on the outside world and a DTO that this component uses to ferry info between the clients and the DAOs. Implementations will constrain the data types they support.

UserConstants
This interface defines the supported profile types and their attributes. It also specifies the mapping of these to database tables and columns. This table is referenced when making the translation of search criteria to database tables in the UserProfileDAO, and when translating between the UserProfile and UserProfileDTO in the UserProfileTranslator.

1.5.2 com.orpheus.user.persistence.ejb

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

PendingConfirmationHomeLocal
This is the local home interface for managing pending confirmations. The local client will obtain it to get the local interface.

PendingConfirmationLocal
This is the local interface used to talk to the PendingConfirmationBean. Supports all client operations.

PendingConfirmationHomeRemote
This is the remote home interface for managing pending confirmations. The remote client will obtain it to get the remote interface.

PendingConfirmationRemote
This is the remote interface used to talk to the PendingConfirmationBean. Supports all client operations.

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

ConfirmationMessageDTO
Simple transfer bean that exists parallel to the ConfirmationMessage class 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.

UserProfileHomeLocal
This is the local home interface for managing user profiles. The local client will obtain it to get the local interface.

UserProfileLocal
This is the local interface used to talk to the UserProfileBean. Supports all client operations.

UserProfileHomeRemote
This is the remote home interface for managing user profiles. The remote client will obtain it to get the remote interface.

UserProfileRemote
This is the remote interface used to talk to the UserProfileBean. Supports all client operations.

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

UserProfileDTO
Simple transfer bean that exists parallel to the UserProfile class 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.

1.5.3 com.orpheus.user.persistence.impl

This is the package where the translator and data access implementations, including custom value objects, are located.

ConfirmationMessageTranslator

Implements ObjectTranslator. It translates between the ConfirmationMessage and the ConfirmationMessageDTO. It is plugged into the OrpheusPendingConfirmationStorage class to perform these translations. The translation is a simple 1-1 mapping between these two entities.

UserProfileTranslator
Implements ObjectTranslator. It translates between the UserProfile and the UserProfileDTO. It is plugged into the OrpheusUserProfilePersistence class to perform these translations. The translation is somewhat sophisticated as it involves the use of the ConfigProfileTypeFactory to get the necessary profile types, and the matching of profile types and the bean object model presented in this package. This mapping is one bean to many profile types 1-1. Usually 1 bean to 3 profile types:

Player <-> “player”, “credentials”, and “base” profile types
Admin <-> “admin”, “credentials”, and “base” profile types
Sponsor <-> “sponsor”, “credentials”, and “base” profile types
ContactInfo <-> “address”, and “base” profile types

In general, a person will be either of the three types of users.

In order to be able to configure itself, it obtains a namespace that it plugs into the ConfigProfileTypeFactory so the latter can configure its supported profile types.

SQLServerPendingConfirmationDAO
Implements PendingConfirmationDAO. Works with SQL Server database and the pending_email_conf table. The mapping is 1-1. It supports all defined CRUD operations. It also supports caching messages to minimize SQL traffic. It uses Config Manager 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 ConfrmationMessageDTO objects.

SQLServerUserProfileDAO
Implements UserProfileDAO. Works with SQL Server database and the player, admin, sponsor, user, and contact_info tables. The mapping is 1-1. It supports all defined CRUD operations. It also supports caching profiles to minimize SQL traffic. It uses Config Manager 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 UserProfileDTO objects.

User
Simple Serializable bean that represents the information in the user table for a user profile. This object will be wrapped inside a UserProfileDTO and transported between the client and the data access object across the EJB transport layer. It

represents some common information to all users, such as handle name, email, password, and active status. A User will always be either a Player, Admin, or Sponsor.

Player
Simple Serializable bean that represents the information in the player and user table for a user profile. As such, it extends the User class. This object will be wrapped inside a UserProfileDTO and transported between the client and the data access object across the EJB transport layer. In addition to common user information, it specifies the payment preferences. It contains info for the “player”, “credentials”, and “base” profile types.

Admin
Simple Serializable bean that represents the information in the admin and user table for a user profile. As such, it extends the User class. This object will be wrapped inside a UserProfileDTO and transported between the client and the data access object across the EJB transport layer. The admin does not define any additional info, so it’s just a marker bean for an “admin” profile type. It contains info for the “admin”, “credentials”, and “base” profile types.

Sponsor
Simple Serializable bean that represents the information in the sponsor and user table for a user profile. As such, it extends the User class. This object will be wrapped inside a UserProfileDTO and transported between the client and the data access object across the EJB transport layer. In addition to common user information, it specifies the sponsor fax, payment preferences, and approval status (which can be null, true, or false). It contains info for the “sponsor”, “credentials”, and “base” profile types.

ContactInfo
Simple Serializable bean that represents the information in the contact_info table for a user profile. This object will be wrapped inside a UserProfileDTO and transported between the client and the data access object across the EJB transport layer. It specifies additional contact info not available in the “base” profile type, such as address, city, state, postal code, and telephone number . It contains info the “address” and “base” profile types.

1.6 Component Exception Definitions

This component defines six custom exceptions.

UserPersistenceException
This exception is the base exception for this component. It extends BaseException.

ObjectInstantiationException
This exception is thrown by the constructors of most custom classes in this design that require configuration. The exception, no pun intended, is the

OrpheusUserProfilePersistence, which uses an exception from another component. The classes that use this exception include: OrpheusPendingConfirmationStorage, LocalOrpheusPendingConfirmationStorage, RemoteOrpheusPendingConfirmationStorage, PendingConfirmationDAO, UserProfileDAO, and UserProfileTranslator. It is thrown if there is an error during the construction of these objects.

PersistenceException
Extends UserPersistenceException. This exception is the base exception for persistence operations in this component. As such, it, and its two 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. Both DAOs, 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. Both DAOs, and the associated EJB and client helper methods, use it.

TranslationException
Extends UserPersistenceException. This exception is thrown by ObjectTranslator and its subclasses if there is an error while doing the translations. Only the UserProfileTranslator implementation uses it since it needs to catch any exceptions thrown while obtaining user types while assembling the UserProfile from the data transfer object.

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 and their constituents are not, and neither is UserProfile, 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.

All caches, DAOs, clients, and translators are thread-safe inasmuch as they hold no mutable state. The DAOFActory 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 UserProfile 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 their constituent classes thread-safe. But above all, the UserProfile class would have to be at least used in some thread-safe wrapper.

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.
- Base Exception 1.0
    - TopCoder standard for all custom exceptions.
- 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 messages and profiles for efficiency.
- Logging Wrapper 1.2
    - Used by clients to log errors.
- User Profile 1.0
    - Provides the UserProfile and ProfileType entities.
- User Profile Manager 1.0
    - Provides the UserProfilePersistence and ConfigProfileTypeFactory entities. This will be the main user of this class, via its manager.
- Email Confirmation 1.0
    - Provides the PendingConfirmationStorageInterface and ConfirmationMessage entities.
- 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.user.persistence
com.orpheus.user.persistence.ejb
com.orpheus.user.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 ObjectTranslator to pass to ObjectFactory. Required	Valid key
cacheKey	Key for the Cache to pass to ObjectFactory. Required	Valid key
log	Name of log	Valid log name
jndiEjbReference	The JNDI reference for the EJB. Required	Example: "java:comp/env/ejb/UserProfileLocal"

3.2.2 DAOFactory

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
pendingConfirmationDAO	Key for the PendingConfirmationDAO to pass to ObjectFactory. Required	Valid key
userProfileDAO	Key for the UserProfileDAO to pass to ObjectFactory. Required	Valid key

3.2.3 DAO implementations

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

3.2.4 UserProfileTranslator

Parameter	Description	Sample Values
specNamespace	Namespace to use with the ConfigManagerSpecificationFactory. Required	Example: “com.topcoder.specify”
factoryKey	Key for the ConfigProfileTypeFactory 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

These are provided in the tables.sql file in the /docs/mappings directory.

3.3.3 EJB deployment descriptor

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

3.3.4 User Profile Types configuration

These are provided in the ProfileTypes.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 before the JVM is started up.

4.3 Demo

See 4.2 above on the required steps to set up the environment. In general, the demo for this class can be just copied from the User Profile Manager and Email Confirmation, so any demo here will not introduce anything new besides showing the vanilla persistence methods in action. As such, the demo will be brief.

4.3.1 Typical Pending Confirmation client usage

This demo will demonstrate typical usage of the OrpheusPendingEmailStorage 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 some kind of manager, but we will show method calls directly.

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

// we might begin by inserting some new messages into persistence
```

```
ConfirmationMessage mess1 = confirmation message with address
"me1@tc.com"
ConfirmationMessage mess2 = confirmation message with address
"me2@tc.com"
client.store(mess1);
client.store(mess2);
// at this point, both are inserted and cached at both levels

// we retrieve a cached and non-cached entity
ConfirmationMessage cMess1 = client.retrieve("me1@tc.com");
ConfirmationMessage cMess3 = client.retrieve("me3@tc.com");
// "me3@tc.com" comes from DB since not in cache. After this retrieval it
// is cached

// we now test if a confirmation message exists for a given address
String existingAddress = "inthere@tc.com";
String nonExistingAddress = "notthere@tc.com";
boolean result1 = client.contains(existingAddress); // true
boolean result2 = client.contains(nonExistingAddress); // false
// both caches are updated with the existing address's confirmation
// message

// we want to delete a confirmation message
client.delete("me1@tc.com");
// user deleted from persistence and both caches

// retrieve all addresses: all will come directly from database
Enumeration allAddresses = client.getAddresses();
// the client caches/recaches all of these messages
```

4.3.2 *Typical User Profile client usage*

This demo will demonstrate typical usage of the OrpheusUserProfilePersistence 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
OrpheusUserProfilePersistence client = new OrpheusUserProfilePersistence
(“myNamespace”);

// we might begin by inserting some new users into persistence
UserProfile user1 = some new user with id 1
UserProfile user2 = another new user with id 2
client.insertProfile(user1);
client.insertProfile(user2);
// at this point, both are inserted and cached at both levels

// we retrieve a cached and non-cached entity
UserProfile cUser1 = client.retrieveProfile(1);
UserProfile cUser3 = client.retrieveProfile(3);
// user 3 comes from DB since not in cache. After this retrieval it is
// cached

// we now update a profile, after we make some changes (not shown)
client.updateProfile(cUser3);
// both caches are updated with this

// we want to delete a user
client.deleteProfile(1);
// user deleted from persistence and both caches
```

```
// retrieve all sponsors that are approved: all will come directly from
// database
Map criteria = map with single criterion for getting all approved
sponsors
UserProfile[] approvedSponsors = client.findProfiles(criteria);
// the client caches all of these sponsors

// retrieve all users: all will come directly from database
UserProfile[] allProfiles = client.retrieveAllProfiles();
// the client caches/recaches all of these profiles

// commit is not used as it does not fit into the usage context (RS
// 1.2.7)
```

5. Future Enhancements

1) If additional user properties are devised for a future version of the application then this component will be upgraded to handle them. This will be made much easier by decoupling data from the transfer path. User profile data can be changed by the introduction of client and DAO implementations without changing any EJB, or even the abstract clients.
2) It may also be upgraded to remove direct access to and storage of plain-text user passwords in favor of encrypted passwords.