IM Persistence 1.0 Component Specification

1. Design

The IM Persistence component provides database implementations of the persistence interfaces for a chat application, including user profiles, roles, categories and statuses. For user profiles, two implementations are provided. One is for registered users, the other for unregistered users. Their database schemas are different.

This component is in fact, a compilation of the four separate parts mentioned above:

1. Profile key manager implementation
2. Two chat user profile persistence implementations, for registered and unregistered users.
3. Role and category persistence
4. Status persistence

All implemented creation and update methods will be enhanced to support pluggable validation. This will give these methods the ability to screen data before calls to the database are made. This functionality is provided by the Data Validation component. All implementations will also make extensive use of the Config Manager, Object Factory, DB Connection Factory, and ID Generator TopCoder components to perform their tasks. All primary keys will be assigned by the ID Generator component.

The following subsection provides an overview of each part of this component.

1. Profile key manager implementation

This will be a simple implementation of the *ProfileKeyManager* interface from the Chat User Profile v2.0 component that is backed by an Informix 10 database. It supports pluggable validation in its createProfileKey method.

2. Chat user profile persistence implementations

There are two implementations: One for registered, and one for unregistered users. These are realized with *RegisteredChatUserProfileInformixPersistence* and *UnregisteredChatUserProfileInformixPersistence*, respectively. Both implement *ChatUserProfilePersistence*. Both mostly implement only the retrieval methods, except for the *UnregisteredChatUserProfileInformixPersistence*, which also creates chat user profiles. *UnregisteredChatUserProfileInformixPersistence* also supports pluggable validation in that method.

3. Role and category persistence

This design goes beyond the requirements and provide an interface that defines the methods, and the persistence extension. Please note that the update method's purpose and expected behavior is still not resolved. The interface provided is

RoleCategoryPersistence, and the implementation is *InformixRoleCategoryPersistence*.

4. Status persistence

This is a simple implementation of the *EntityStatusTracker* interface backed by Informix 10 database.

1.1 Design Patterns

DAO: All implementations act as data access objects, encapsulating access to a database.

Strategy: Not actually used in this design.

1.2 Industry Standards

JDBC, SQL

1.3 Required Algorithms

None. The developer is encouraged to optimize all SQL operations. The methods describe these in detail, but the developer is expected to not only provide the actual SQL code, but make is as efficient as possible.

1.4 Component Class Overview

InformixProfileKeyManager:

This *ProfileKeyManager* implementation performs *ProfileKey* persistence with an Informix 10 database. It implements all methods. It makes use of the Config Manager, ObjectFactory, ID Generator, Data Validation, and DB Connection Factory components.

It contains three constructors. One takes a namespace that is used to obtain configuration parameters from Config Manager. These configuration parameters are used then to create instances of the required *DBConnectionFactory* and the optional *ObjectValidator*. Configuration also supplies the connection name to be used with the connection factory to obtain connections for all methods. Lastly, the id generation name, also obtained from configuration, is used to obtain the *IDGenerator* instance from the *IDGeneratorFactory*. The default constructor does the same thing but used a default namespace. The third constructor simply takes all these pieces as parameters and sets them to its fields.

This manager works with the *all_user* table. The *ProfileKey* values are mapped to the table in the following manner: ‘id’ attribute maps to *user_id* column; ‘username’ attribute maps to *username* column, and ‘type’ attribute maps to *registered_flag* column. The *create_date* and *modify_date* will be set by this class to the date and time of the operation (in this case, only the createProfileKey method, since it is the only method that modifies the table and leaves a row afterwards). The *create_user* and *modify_user* are set to the current user in Informix.

Validation, as represented by the use of the Data Validation component, is only used in the createProfileKey method. It can be plugged into the method to provide additional validation of the passed *ProfileKey*, in addition to the already present check for a legal type ("Registered" and "Unregistered"). Some suggestions might be checking for valid or restricted usernames.

RegisteredChatUserProfileInformixPersistence:
This *ChatUserProfilePersistence* implementation performs registered *ChatUserProfile* persistence with an Informix 10 database. Specifically, it only supports the retrieval and searching methods. Creation, updating, and deletion are not supported. It makes use of the Config Manager, ObjectFactory and DB Connection Factory components.

It contains three constructors. One takes a namespace that is used to obtain configuration parameters from Config Manager. These configuration parameters are used then to create instances of the required *DBConnectionFactory*. Configuration also supplies the connection name to be used with the connection factory to obtain connections for all methods. Lastly, the search method requires a mapping of user-defined attribute names to actual column names. A static set of user-defined attribute names are provided in the *UserDefinedAttributeNames* class, and the column names are provided in this class.

For this class, this is the mapping (*ChatUserProfile* **è** Informix column name)
- "First Name" **è** "first_name"
- "Last Name" **è** "last_name"
- "Company" **è** "company_name"
- "Title" **è** "title"
- "Email" **è** "address"
- "Name" **è** n/a (This is simply put into the *ChatUserProfile* when it is returned. It will just be the username.).

The default constructor does the same thing but used a default namespace. The third constructor simply takes all these pieces as parameters and sets them to its fields, with the same action regarding the mappings as described above with the namespace constructor.

This persistence implementation works with many tables. The central table is *user*, with the first and last names, as well as the *handle* column that represents the username, which will be unique. When retrieving a *ChatUserProfile*, one record from this table will be returned. The *user* table links, by *user_id*, to the email table, from which it obtains the email *address*. The relationship to a company more complex, as a user and company are linked by an intermediate table. Therefore, it is possible that a user will have many links to the *company* table. For the purposes of this persistence, only one such link is required. There is no preference as to which one, so a simple query to get one will be sufficient. The *contact* table is the link table between *user* and *contact*, by *user_id* and

contact_id, respectively. The *contact* table contains the *title* column, and the *company_id* would be linked back to the *company* table to obtain the *company_name* column.

UnregisteredChatUserProfileInformixPersistence:
This *ChatUserProfilePersistence* implementation performs unregistered *ChatUserProfile* persistence with an Informix 10 database. Specifically, it only supports the creation, retrieval and searching methods. Updating, and deletion are not supported. It makes use of the Config Manager, ObjectFactory, ID Generator, Data Validation, and DB Connection Factory components.

It contains three constructors. One takes a namespace that is used to obtain configuration parameters from Config Manager. These configuration parameters are used then to create instances of the required *DBConnectionFactory* and the optional *ObjectValidator*. Configuration also supplies the connection name to be used with the connection factory to obtain connections for all methods. The id generation name, also obtained from configuration, is used to obtain the *IDGenerator* instance from the *IDGeneratorFactory*. Lastly, the search method requires a mapping of user-defined attribute names to actual column names. A static set of user-defined attribute names are provided in the *UserDefinedAttributeNames* class, and the column names are provided in this class.

For this class, this is the mapping (*ChatUserProfile* **è** Informix column name)
- "First Name" **è** "first_name"
- "Last Name" **è** "last_name"
- "Company" **è** "company"
- "Title" **è** "title"
- "Email" **è** "email"
- "Name" **è** n/a (This is simply put into the *ChatUserProfile* when it is returned. It will be the concatenation of the First Name and Last Name).

The default constructor does the same thing but used a default namespace. The third constructor simply takes all these pieces as parameters and sets them to its fields, with the same action regarding the mappings as described above with the namespace constructor.

This persistence implementation works with the *client* table. The *ChatUserProfile* "id' and 'username" values are mapped to the *client_id* column, with the other user-defined attributes matching the column names. The *create_date* and *modify_date* will be set by this class to the date and time of the operation (in this case, only the createProfileKey method, since it is the only method that modifies the table and leaves a row afterwards). The *create_user* and *modify_user* are set to the current user in Informix.

Validation, as represented by the use of the Data Validation component, is only used in the createProfile method. It can be plugged into the method to provide additional validation of the passed *ChatUserProfile*. The *NullChatUserProfileAttributeValidator* is one such validator provided. It simply makes sure that none of the required user-defined attributes are null: First name, last name, and email.

NullChatUserProfileAttributeValidator:
An *ObjctValidator* implementation that verifies that a *ChatUserProfile* has non-null and non-empty first name, last name, and email attributes defined. It is meant to be used in the *UnregisteredChatUserProfileInformixPersistence* class via configuration. It is not provided by default, so it must be set explicitly, which would be preferable since the Informix database needs them to be available.

UserDefinedAttributeNames:
A simple listing of user-defined attribute names to be used as keys in the *ChatUserProfile* class. These include first name, last name, company name, title, email, and name. These will be used as names in the *ChatUserProfile* class to be mapped in the *RegisteredChatUserProfileInformixPersistence* and *UnregisteredChatUserProfileInformixPersistence* classes to configured column names in the Informix database.

Category:
This is an object representation of the user in the Role-Category persistence. It matches the *principal* table. It will be created in the *RoleCategoryPersistence* implementation.

User:
This is an object representation of the category in the Role-Category persistence. It matches the *category* table. It will be created in the *RoleCategoryPersistence* implementation.

RoleCategoryPersistence:
This interface defines the contract for performing user role and category retrievals as well as category updates. There are three methods to retrieve *Categories*, and one to retrieve *Users* by roles. There is also a method to update the associations of categories to managers.

InformixRoleCategoryPersistence:
This *RoleCategoryPersistence* implementation performs user role and category persistence with an Informix 10 database. It makes use of the Config Manager, ObjectFactory, Data Validation, and DB Connection Factory components.

It contains three constructors. One takes a namespace that is used to obtain configuration parameters from Config Manager. These configuration parameters

are used then to create instances of the required *DBConnectionFactory* and the optional *ObjectValidator*. Configuration also supplies the connection name to be used with the connection factory to obtain connections for all methods.

The default constructor does the same thing but used a default namespace. The third constructor simply takes all these pieces as parameters and sets them to its fields.

This persistence works with the *principal*, *role*, and *principal_role* tables for the user operation, and the *principal*, *category*, and *manager_category* tables for categories. For users, when retrieving users for a role, the link table *principal_role* is used for determining which users (principals) are related to what roles. Similarly, then managing associations between a manager and a category, the link table manager_category is queried to determine existing associations between categories and managers and to create new ones. Note that in this context managers are users managed in the principal table. The create_date and modify_date will be set by this class to the date and time of the operation (in this case, only the updateCategories method, since it is the only method that modifies the table and leaves a row afterwards). The create_user and modify_user are set to the current user in Informix.

Validation, as represented by the use of the Data Validation component, is only used in the updateCategories method. It can be plugged into the method to provide additional validation of the passed *Category* array.

InformixEntityStatusTracker:
This *EntityStatusTracker* implementation performs status persistence with an Informix 10 database. It implements all methods. It makes use of the Config Manager, ObjectFactory, and DB Connection Factory components.

It contains three constructors. One takes a namespace that is used to obtain configuration parameters from Config Manager. These configuration parameters are used then to create instances of the required *DBConnectionFactory*. Configuration also supplies the connection name to be used with the connection factory to obtain connections for all methods. The default constructor does the same thing but used a default namespace. The third constructor simply takes all these pieces as parameters and sets them to its fields.

This manager works with the xxx_status and xxx_status_history tables, where xxx represents the status entity type (Entity.type field). The *create_date* and *modify_date* will be set by this class to the date and time of the operation (in this case, only the setStatus method, since it is the only method that modifies the table and leaves a row afterwards). The *create_user* and *modify_user* are set to the current user in Informix.

1.5 Component Exception Definitions

This design creates nine custom exceptions.

ConfigurationException:
The implementations will throw this exception if they encounter any issue during its construction in the *ConfigManager*-backed constructors. Conditions include the passed namespace being unrecognized, the *ObjectFactory* not being configured for creating a *DBConnectionFactory* or *ObjectValidator* instance, or *IDGeneratorFactory* not being configured for the asked *IDGenerator*. Extends BaseException.

ProfileKeyValidationException:
This exception indicates that a passed *ProfileKey* in the *InformixProfileKeyManager* createProfileKey method has failed validation with a configured *ObjectValidator*. To fit into the existing exception hierarchy, it extends *ProfileKeyManagerPersistenceException*.

ChatUserProfileValidationException:
This exception indicates that a passed *ChatUserProfile* in the *UnregisteredChatUserProfileInformixPersistence* createProfile method has failed validation with a configured *ObjectValidator*. To fit into the existing exception hierarchy, it extends *ChatUserProfilePersistenceException*.

RoleCategoryPersistenceException:
This exception indicates that there is a general error during the persistence operation in *RoleCategoryPersistence.* It extends BaseException.

RoleNotFoundException:
This exception indicates that an indicated role does not exist in persistence. It is thrown in the *RoleCategoryPersistence* getUsers method. It extends *RoleCategoryPersistenceException*.

ManagerNotFoundException:
This exception indicates that an indicated manager does not exist in persistence. It is thrown in the *RoleCategoryPersistence* getCategories(String) and updateCategories methods. It extends *RoleCategoryPersistenceException*.

CategoryValidationException:
This exception indicates that a passed *Category* in the *InformixRoleCategoryPersistence* updateCategories method has failed validation with a configured *ObjectValidator*. It extends *RoleCategoryPersistenceException*.

EntityStatusValidationException:
This exception indicates that a passed *EntityKey* in the *InformixEntityStatusTracker* setStatus method has failed validation with a

configured *ObjectValidator*. To fit into the existing exception hierarchy, it extends *PersistenceException* from the Status Tracker component.

CategoryNotFoundException:
This exception indicates that an indicated category does not exist in persistence. It is thrown in the RoleCategoryPersistence updateCategories method. It extends RoleCategoryPersistenceException.

1.6 Thread Safety

The component is effectively thread-safe. All implementations are immutable, and it is expected that the beans passed to the objects will be manipulated by only one thread at a time, so concurrent usage of a bean instance is not an issue.

All write operations are also transactional. Any error in a write operation will result in the whole operation being rolled back. Note that in some cases, an operation involves just one write action, and thus does not require explicit transaction declaration. In such cases, the method relies on default transactional control (for example, see method *InformixProfileKeyManager.createProfileKey*).

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.4 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- Config Manager 2.1.5
  - Used to maintain configuration properties for all persistence implementations.
- Data Validation 1.0
  - Provides the validation framework for all persistence implementations.
- DB Connection Factory 1.0
  - Provides an easy API for obtaining JDBC Connections for all persistence implementations.
- Object Factory 2.0.1
  - Provides easy object instantiation APIs for all persistence implementations.
- ID Generator 3.0
  - Provides unique ID generation for the profile key manager, unregistered chat user profile persistence, and
- Chat User Profile 2.0
  - Provides the chat user profile framework into which the profile key manager and chat user profile persistence implementations plug into.
- Status Tracker 1.0
  - Provides the *EntityStatusTracker* interface that this component extends. This is the correct component to mention, and not Chat Status Tracker (which, however, will still use this implementation).

- Base Exception 1.0
    - Provides common base exception for all TC components, and is a standard for all TC components.

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

None

3. Installation and Configuration

3.1 Package Name

com.cronos.im.persistence
com.cronos.im.persistence.rolecategories

3.2 Configuration Parameters

3.2.1 InformixProfileKeyManager

Parameter	Description	Values
specification_factory_namespace	The namespace to initialize the ConfigManagerSpecificationFactory with. REQUIRED	Any valid configuration namespace
db_connection_factory_key	The key to put into ObjectFactory to obtain an instance of DBConnectionFactory. REQUIRED	Any unique key is fine. Must be configured in ObjectFactory.
connection_name	A name of a connection to use. REQUIRED	Any unique name is fine. Must be configured in DBConnectionFactory.
validator_key	The key to put into ObjectFactory to obtain an instance of ObjectValidator. OPTIONAL	Any unique key is fine. Must be configured in ObjectFactory.
id_generator	A name of an IDGenerator to obtain from IDGeneratorFactory. REQUIRED	Any unique name is fine. Must be configured in IDGeneratorFactory.

3.2.2 RegisteredChatUserProfileInformixPersistence

Parameter	Description	Values
specification_factory_namespace	The namespace to initialize the ConfigManagerSpecificationFactory with. REQUIRED	Any valid configuration namespace
db_connection_factory_key	The key to put into ObjectFactory to obtain an instance of DBConnectionFactory. REQUIRED	Any unique key is fine. Must be configured in ObjectFactory.
connection_name	A name of a connection to use. REQUIRED	Any unique name is fine. Must be configured in DBConnectionFactory.

3.2.3 InformixRoleCategoryPersistence, UnregisteredChatUserProfileInformixPersistence

Parameter	Description	Values
specification_factory_namespace	The namespace to initialize the ConfigManagerSpecificationFactory with. REQUIRED	Any valid configuration namespace
db_connection_factory_key	The key to put into ObjectFactory to obtain an instance of DBConnectionFactory. REQUIRED	Any unique key is fine. Must be configured in ObjectFactory.
connection_name	A name of a connection to use. REQUIRED	Any unique name is fine. Must be configured in DBConnectionFactory.
validator_key	The key to put into ObjectFactory to obtain an instance of ObjectValidator. OPTIONAL	Any unique key is fine. Must be configured in ObjectFactory.

3.2.4 InformixEntityStatusTracker

Parameter	Description	Values
specification_factory_namespace	The namespace to initialize the ConfigManagerSpecificationFactory with. REQUIRED	Any valid configuration namespace
db_connection_factory_key	The key to put into ObjectFactory to obtain an instance of DBConnectionFactory. REQUIRED	Any unique key is fine. Must be configured in ObjectFactory.
connection_name	A name of a connection to use. REQUIRED	Any unique name is fine. Must be configured in DBConnectionFactory.
validator_key	The key to put into ObjectFactory to obtain an instance of ObjectValidator. OPTIONAL	Any unique key is fine. Must be configured in ObjectFactory.

3.3 Dependencies Configuration

The developer should look at the documentation of the components used in this component (defined in section 2.2). In the demo, sample configuration of the ConfigManager and ObjectFactory will be provided.

The DDL can be obtained from the requirements document. It is an acceptable practice to refer directly to the requirements specifications for information.

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

All external components that are used by this component will need to have their jars added to the class path and be configured, and the Informix database will need to be initialized with relevant tables.

4.3 Demo

The demo, following the pattern set out by this specification, splits the demo into four sections. Each section, in turn, will provide two subsections: The first to set up the demo, and the second to perform a scenario.

4.3.1 Informix Profile Key Manager

4.3.1.1 Setup

This subsection details the setup of the *InformixProfileKeyManager*.

First, we define a sample *ObjectValidator* implementation that will screen for some obscene usernames. It is expected that in the real world, such validation would occur at the business layer, so this is just shown for illustrative purposes.

```
Package com.myspace.validators;

import com.topcoder.chat.user.profile.*;

public class ObsceneUsernameValidator {

    // other parts omitted for clarity

    public String getMessage(java.lang.Object obj) {
        ProfileKey key = (ProfileKey) obj;
        String username = key.getUsername();
        if ("Fracker".equals(username) || "DirtEater".equals(username) ||
"Scumbag".equals(username)) {
            // username is obscene
            return username + " is an obscene name. Please use another";
        }

        // username is fine
        return null;
    }
}
```

Here we define the configuration parameters in *ConfigManager*:

```
<Config name="TestConfig">
   <Property name="specification_factory_namespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="db_connection_factory_key">
         <Value>connectionFactoryKey</Value>
   </Property>
   <Property name="connection_name">
         <Value>profileKeyConnection</Value>
   </Property>
   <Property name="validator">
         <Value>obsceneValidatorKey</Value>
```

```
   </Property>
   <Property name="id_generator">
         <Value>profileKey</Value>
   </Property>
</Config>
```

Here we define the configuration parameters in *ObjectFactory*:

```
<Config name="SpecificationNamespace ">

<Property name="connectionFactoryKey">
   <Property name="type">
      <Value> com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
      </Value>
   </Property>
   <Property name="params">
      <Property name="param1">
         <Property name="type">
            <Value>String</Value>
         </Property>
         <Property name="value">
            <Value>ConnectionFactoryNamespace</Value>
         </Property>
      </Property>
   </Property>
</Property>

<Property name="obsceneValidatorKey">
   <Property name="type">
      <Value>com.myspace.validators.ObsceneUsernameValidator</Value>
   </Property>
</Property>

</Config>
```

The configuration of DB Connection Factory and ID Generator components are left to the developer, but would contain configuration for an ID Generator named "profileKey" and a DB Connection Factory namespace "ConnectionFactoryNamespace" to mesh with the sample configuration provided above.

Finally, we can prepare the following data in the *all_user* table. The table below will contain several existing profile keys. For clarity only the relevant columns *user_id*, *registered_flag*, and *username* are shown. The *create_XXX* and *modify_XXX* are not shown.

Table *all_user*:

user_id	registered_flag	username
1	Y	evirn
2	N	AleaActaEst
3	N	argolite
4	Y	Wendell

The data above means that users “argolite” and “AleaActaEst” are of type “Unregistered”, whereas users “evirn” and “Wendell” are “Registered”.

4.3.1.2 Typical usage scenario

This subsection details the use of the *InformixProfileKeyManager* . A typical scenario will involve using all methods.

```
// instantiate the component with the namespace constructor
ProfileKeyManager manager = new InformixProfileKeyManager("TestConfig");

// the administration may begin with creating new profiles, and
// attempting to add them to the manager.
ProfileKey userA = new ProfileKey(“drake”,”Registered”);
ProfileKey userB = new ProfileKey(“jonah”,”Unregistered”);
ProfileKey userC = new ProfileKey(“Scumbag”,”Registered”);
ProfileKey userD = new ProfileKey(“yearner”,”Wrong”);

ProfileKey userAA = manager.createProfileKey(userA);
ProfileKey userBB = manager.createProfileKey(userB);
ProfileKey userCC = manager.createProfileKey(userC);
ProfileKey userDD = manager.createProfileKey(userD);

// userA and userB would be added. userC and userD would not be added
// because userC contains an obscene word that is banned by the
// configured validator, and userD contains an illegal type. Note that
// exception handling is omitted above for clarity. The all_user table
// would then appear as follows (assuming no gaps in the ID generation)
// with new entries with blue highlight.
// Please note the registered_flag column’s new values. Also note that
// the return profile keys would contain the generated user ids.
```

Table *all_user (post-creation)*:

user_id	registered_flag	username
1	Y	evirn
2	N	AleaActaEst
3	N	argolite
4	Y	Wendell
5	Y	drake
6	N	jonah

```
// further administration would involve removing some users. In this
// case, we wish to remove user “evirn”.
manager.deleteProfileKey(1);

// The all_user table would then appear as follows:
```

Table *all_user (post-deletion)*:

user_id	registered_flag	username
2	N	AleaActaEst
3	N	argolite
4	Y	Wendell
5	Y	drake
6	N	jonah

```
// As part of some front-end application usage, the profile key table can
// be queried. All queries would be performed against the all_user
// table in the above state. Note that the resulting profiles will be
// referred to by their usernames: For example, jonah will refer to the
// profile key with username "jonah".

// a query using a user ID
ProfileKey userQ11 = manager.getProfileKey(2); // retrieves AleaActaEst
ProfileKey userQ12 = manager.getProfileKey(1); // retrieves null
// The getProfileKey(String,String) method would work in the same manner

// a query using multiple user IDs
long[] userIDs = {1,2,3};
ProfileKey[] usersQ21 = manager.getProfileKeys(userIDs);
// usersQ21 will have length of 3, just like the input, and contain null,
// AleaActaEst, and argolite profiles
// The getProfileKeys(String[],String) method would work in the same
// manner

// a query using a type
ProfileKey[] usersQ31 = manager.getProfileKeys("Registered");
// usersQ21 will have length of 2, representing the two registered users
// in the table: Wendell and drake.
// The getProfileKeys(String[]) method would work in the same
// manner
```

4.3.2 Chat User Profile Persistence Implementations

4.3.2.1 RegisteredChatUserProfileInformixPersistence setup

This subsection details the setup of the *RegisteredChatUserProfileInformixPersistence*.

Here we define the configuration parameters in *ConfigManager*:

```
<Config name="TestConfig">
   <Property name="specification_factory_namespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="db_connection_factory_key">
         <Value>connectionFactoryKey</Value>
   </Property>
   <Property name="connection_name">
         <Value>profileConnection</Value>
   </Property>
</Config>
```

Here we define the configuration parameters in *ObjectFactory*:

```
<Config name="SpecificationNamespace ">

<Property name="connectionFactoryKey">
   <Property name="type">
      <Value> com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
      </Value>
   </Property>
```

```
   <Property name="params">
      <Property name="param1">
         <Property name="type">
            <Value>String</Value>
         </Property>
         <Property name="value">
            <Value>ConnectionFactoryNamespace</Value>
         </Property>
      </Property>
   </Property>
</Property>

</Config>
```

The configuration of the DB Connection Factory and ID Generator component is left to the developer, but would contain a DB Connection Factory namespace "ConnectionFactoryNamespace" to mesh with the sample configuration provided above.

Finally, we can prepare the following data in all tables for the retrieval demonstration. Note that the contact table defines two contacts for a user. The effects of this will be shown in the subsequent demo.

Table *user*:

user_id	first_name	last_name	handle
1	Ender	Virn	evirn
2	Jim	Mann	jmann
3	Chris	Rack	crack

Table *email*:

user_id	address
1	evirn@bread.com
2	jmann@bread.com
3	crack@bread.com

Table *company*:

company_id	company_name
1	MicroSoft
2	TopCoder
3	Sun
4	Nabisco

Table *contact*:

user_id	company_id	title
1	1	coder
2	2	designer
3	3	architect
3	4	Project Manager

4.3.2.2 Typical RegisteredChatUserProfileInformixPersistence usage scenario

```
// instantiate the component with the namespace constructor
ChatUserProfilePersistence persistence = new
RegisteredChatUserProfileInformixPersistence("TestConfig");

// This persistence implementation only supports retrieval
// the two getter methods can be used to access info about users

// a query using an existing username
ChatUserProfile profile = persistence.getProfile(“evirn”);
// retrieves evirn data, including
// - First Name = Ender
// - Last Name = Virn
// - Company = MicroSoft
// - Title = coder
// - Email = evirn@bread.com
// - Name = evirn

// a query using several usernames: evirn, blah, crack
String[] usernames = {“evirn”,”blah”,”crack”};
ChatUserProfile[] profiles = persistence.getProfiles(usernames);
// retrieves evirn data in first spot, including
// - First Name = Ender
// - Last Name = Virn
// - Company = MicroSoft
// - Title = coder
// - Email = evirn@bread.com
// - Name = evirn
// retrieves null in second spot
// retrieves crack data in last spot, including
// - First Name = Chris
// - Last Name = Rack
// - Company = Sun
// - Title = architect
// - Email = crack@bread.com
// - Name = crack
// Note that for user crack, we retrieved only one company and title. No
// special ordering was used to select this company.

// The searches can be quite complex. Here’s a simple example of
// finding all employed in Nabisco company. We will restrict this to the
// given users: evrin, crack, jmann
Map criteria = new HashMap();
String[] registered = {“evirn”,”jmann”,”crack”};
criteria.put(“Company”,”Nabisco”);
ChatUserProfile[] profiles2 = persistence.searchProfiles(criteria,
registered);
// retrieves crack data in only spot, including
// - First Name = Chris
// - Last Name = Rack
// - Company = nabisco
// - Title = Project Manager
// - Email = crack@bread.com
// - Name = crack

// A more complex search example would involve searching on multiple
// items in a category. We reuse the array of registered users.
List titles = new ArrayList();
titles.add(“coder”);
titles.add(“architect”);
```

```
criteria.clear();
criteria.put("Title"," titles");
ChatUserProfile[] profiles3 = persistence.searchProfiles(criteria,
registered);
// retrieves evirn data in first spot, including
// - First Name = Ender
// - Last Name = Virn
// - Company = MicroSoft
// - Title = coder
// - Email = evirn@bread.com
// - Name = evirn
// retrieves crack data in second spot, including
// - First Name = Chris
// - Last Name = Rack
// - Company = Sun
// - Title = architect
// - Email = crack@bread.com
// - Name = crack
```

4.3.2.3 UnregisteredChatUserProfileInformixPersistence setup

This subsection details the setup of the *UnregisteredChatUserProfileInformixPersistence*.

Here we define the configuration parameters in *ConfigManager*:

```
<Config name="TestConfig">
   <Property name="specification_factory_namespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="db_connection_factory_key">
         <Value>connectionFactoryKey</Value>
   </Property>
   <Property name="connection_name">
         <Value>profileKeyConnection</Value>
   </Property>
   <Property name="validator">
         <Value>validatorKey</Value>
   </Property>
   <Property name="id_generator">
         <Value>profile</Value>
   </Property>
</Config>
```

Here we define the configuration parameters in *ObjectFactory*:

```
<Config name="SpecificationNamespace ">

<Property name="connectionFactoryKey">
   <Property name="type">
      <Value> com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
      </Value>
   </Property>
   <Property name="params">
      <Property name="param1">
         <Property name="type">
            <Value>String</Value>
         </Property>
         <Property name="value">
```

```
            <Value>ConnectionFactoryNamespace</Value>
         </Property>
      </Property>
   </Property>
</Property>

<Property name="validatorKey">
   <Property name="type">
      <Value> com.cronos.im.persistence.
NullChatUserProfileAttributeValidator</Value>
   </Property>
</Property>

</Config>
```

The configuration of DB Connection Factory and ID Generator components are left to the developer, but would contain configuration for an ID Generator named "profile" and a DB Connection Factory namespace "ConnectionFactoryNamespace" to mesh with the sample configuration provided above.

Finally, we can prepare the following data in the *client* table. The table below will contain several existing profiles. For clarity only the relevant columns *client_id*, *first_name*, *last_name*, *company*, *title*, and *email* are shown. The *create_XXX* and *modify_XXX* are not shown.

Table *client*:

client_id	first_name	last_name	company	title	email
1	Joe	Pesci	Nabisco	coder	jpesci@me.com
2	Kim	John	MicroSoft	coder	kjohn@me.com
3	Alain	Rock	Sun	architect	arock@me.com
4	Mary	Magden	Bits	designer	mmagden@me.com

4.3.2.4 Typical UnregisteredChatUserProfileInformixPersistence usage scenario

```
// instantiate the component with the namespace constructor
ChatUserProfilePersistence persistence = new
UnregisteredChatUserProfileInformixPersistence("TestConfig");

// the administration may begin with creating new profiles, and
// attempting to add them to the manager.
ChatUserProfile profile1 = // new profile for Jen Jones, a TopCoder
Project manager with email jjones@topcoder.com
persistence.createProfile(profile1);
// the new profile is added, with the new row with blue highlight
```

Table *client (post-creation)*:

client_id	first_name	last_name	company	title	email
1	Joe	Pesci	Nabisco	coder	jpesci@me.com
2	Kim	John	MicroSoft	coder	kjohn@me.com
3	Alain	Rock	Sun	architect	arock@me.com
4	Mary	Magden	Bits	designer	mmagden@me.com

5	Jen	Jones	TopCoder	Project Manager	jjones@topcoder.com

```
// retrieval and search operations can also be performed


// a query using an existing username (i.e. client_id)
ChatUserProfile profile = persistence.getProfile("1");
// retrieves data for client 1, including
// - First Name = Joe
// - Last Name = Pesci
// - Company = Nabisco
// - Title = coder
// - Email = jpesci@me.com
// - Name = JoePesci

// a query using several usernames: 1, 7, 5
String[] usernames = {"1","7","5"};
ChatUserProfile[] profiles = persistence.getProfiles(usernames);
// retrieves client 1 data in first spot, including
// - First Name = Joe
// - Last Name = Pesci
// - Company = Nabisco
// - Title = coder
// - Email = jpesci@me.com
// - Name = JoePesci
// retrieves null in second spot
// retrieves client 5 data in last spot, including
// - First Name = Jen
// - Last Name = Jones
// - Company = TopCoder
// - Title = Project Manager
// - Email = jjones@topcoder.com
// - Name = JenJones

// The searches can be quite complex. Here's a simple example of
// finding all employed in Nabisco company. We will restrict this to the
// given users: 1, 2, 3, 4, 5
Map criteria = new HashMap();
String[] registered = {"1","2","3","4","5"};
criteria.put("Company","Nabisco");
ChatUserProfile[] profiles2 = persistence.searchProfiles(criteria,
registered);
// retrieves one row array data for client 1, including
// - First Name = Joe
// - Last Name = Pesci
// - Company = Nabisco
// - Title = coder
// - Email = jpesci@me.com
// - Name = JoePesci


// A more complex search example would involve searching on multiple
// items in a category. We reuse the array of registered users.
List titles = new ArrayList();
titles.add("coder");
titles.add("architect");
criteria.clear();
criteria.put("Title"," titles");
ChatUserProfile[] profiles3 = persistence.searchProfiles(criteria,
registered);
// retrieves data for client 1, including
```

```
// - First Name = Joe
// - Last Name = Pesci
// - Company = Nabisco
// - Title = coder
// - Email = jpesci@me.com
// - Name = JoePesci
// retrieves data for client 2, including
// - First Name = Kim
// - Last Name = John
// - Company = MicorSoft
// - Title = coder
// - Email = kjohn@me.com
// - Name = KimJohn
// retrieves data for client 3, including
// - First Name = Alain
// - Last Name = Rock
// - Company = Sun
// - Title = architect
// - Email = arock@me.com
// - Name = AlainRock
```

4.3.3 InformixRoleCategoryPersistence

4.3.3.1 InformixRoleCategoryPersistence setup

This subsection details the setup of the *InformixRoleCategoryPersistence*.

Here we define the configuration parameters in *ConfigManager*:

```
<Config name="TestConfig">
   <Property name="specification_factory_namespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="db_connection_factory_key">
         <Value>connectionFactoryKey</Value>
   </Property>
   <Property name="connection_name">
         <Value>defaultConnection</Value>
   </Property>
</Config>
```

Here we define the configuration parameters in *ObjectFactory*:

```
<Config name="SpecificationNamespace ">

<Property name="connectionFactoryKey">
   <Property name="type">
      <Value> com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
      </Value>
   </Property>
   <Property name="params">
      <Property name="param1">
         <Property name="type">
            <Value>String</Value>
         </Property>
         <Property name="value">
            <Value>ConnectionFactoryNamespace</Value>
         </Property>
      </Property>
```

```
   </Property>
</Property>

</Config>
```

The configuration of the DB Connection Factory component is left to the developer, but would contain a DB Connection Factory namespace "ConnectionFactoryNamespace" to mesh with the sample configuration provided above.

Finally, we can prepare the following data in all tables for the demonstration. For clarity, only the relevant columns are shown. The *create_XXX* and *modify_XXX* are not shown, where applicable.

Note that here principal, user, and manager is interchangeable.

Table *principal*:

principal_id	principal_name
1	evirn
2	moss
3	cales

Table *role*:

role_id	role_name
1	admin
2	coder

Table *principal_role*:

principal_id	role_id
1	1
1	2
2	1
3	2

Basically, *evirn* is an *admin* and *coder*, *moss* is an *admin*, and *cales* is a *coder*.

Table *category*:

category_id	name	description	chattable_flag
1	coding	Coding category	Y
2	management	Management category	N
3	modeling	UML modeling category	Y

Table *manager_category*:

manager_id	category_id
1	1
1	2
3	1

Basically, *evirn* is associated with *coding* and *management* categories, and *cales* is associated with the *coding* category.

4.3.3.2 Typical InformixRoleCategoryPersistence usage scenario

```
// instantiate the component with the namespace constructor
RoleCategoryPersistence persistence = new InformixRoleCategoryPersistence
("TestConfig");

// This persistence implementation supports many getter functions:

// a query to get all users with a given role
User[] users = persistence.getUsers("coder");
// retrieves users evirn and cales

// a query to get all categories
Category[] allCategories = persistence.getAllCategories();
// retrieves all three categories

// a query to get categories that are chattable
allCategories = persistence.getCategories(true);
// retrieves categories coding and modeling

// a query to get categories for a given manager
allCategories = persistence.getCategories("evirn");
// retrieves categories coding and management

// we can also change the categories for a user
Category[] updateCategories = categories of coding and modeling
persistence.updateCategories("evirn",updateCategories);
// The manager_category table will become as below, with the new
// association for evirn in blue highlight, and the previous association
// of evirn to management is removed.
```

Table *manager_category (post-update)*:

manager_id	category_id
1	1
3	1
1	3

4.3.4 InformixEntityStatusTracker

4.3.4.1 InformixEntityStatusTracker setup

This subsection details the setup of the *InformixEntityStatusTracker*.

Here we define the configuration parameters in *ConfigManager*:

```
<Config name="TestConfig">
```

```
   <Property name="specification_factory_namespace">
         <Value>SpecificationNamespace</Value>
   </Property>
   <Property name="db_connection_factory_key">
         <Value>connectionFactoryKey</Value>
   </Property>
   <Property name="connection_name">
         <Value>defaultConnection</Value>
   </Property>
</Config>
```

Here we define the configuration parameters in *ObjectFactory*:

```
<Config name="SpecificationNamespace ">

<Property name="connectionFactoryKey">
   <Property name="type">
      <Value> com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
      </Value>
   </Property>
   <Property name="params">
      <Property name="param1">
         <Property name="type">
            <Value>String</Value>
         </Property>
         <Property name="value">
            <Value>ConnectionFactoryNamespace</Value>
         </Property>
      </Property>
   </Property>
</Property>

</Config>
```

The configuration of DB Connection Factory components are left to the developer, but would contain configuration for a DB Connection Factory namespace "ConnectionFactoryNamespace" to mesh with the sample configuration provided above.

Finally, we can prepare the following data in all tables for the demonstration. For clarity, only the relevant columns are shown. The *create_XXX* and *modify_XXX* are not shown, where applicable. We will deal with an entity called "entity", which will have an existing entry. As such, there will be the xxx_status and xxx_status_history tables for *entity*.

Table *entity_status*:

entity_status_id	name	description	chattable_flag
1	init	Init status	Y
2	deal	Deal status	N
3	closing	Closing status	Y

Table *entity_status_history*:

entity_status_id	entity_id	start_date	end_date

1	1	March 1, 2007	March 2, 2007
2	1	March 2, 2007	
1	2	March 7, 2007	

This history basically states that entity 1 has gone through *init* status and now is in *deal* status, and entity 2 is in *init* status.

4.3.4.2 Typical InformixEntityStatusTracker usage scenario

```
// instantiate the component with the namespace constructor
EntityStatusTracker persistence = new InformixEntityStatusTracker
("TestConfig");

// This persistence implementation supports many getter functions:

// a query to get current status for user 1
EntityKey instance = for user 1
EntityStatus currStatus = persistence.getCurrentStatus(instance);
// retrieves status of "deal"

// a query to get all statuses for user 1
EntityStatus[] allStatuses = persistence.getStatusHistory(instance);
// retrieves all two statuses: "init" that started on March 1, 2007 and
// ended on March 2, 2007, and "deal" which started on March 2, 2007 and
// is current (no end date).

// a query to get all current entities in "init"
Entity type = for user
Status[] statuses = one status of "init"
EntityStatus[] allInitStatuses = persistence.findByStatus(type,
                                    statuses);
// retrieves one status: "init" that started on March 7, 2007 and
// is for user 2

// we can also set the status of user 1
Status newStatus = status of "closing"
persistence.setStatus(instance,newStatus,"evirn");
// The user_status_history is updated as below.
// There is a new entry for user 1, and the previous current status is
// ended. This can be seen in blue highlight, and the previous
// of evirn to management is removed. Not seen is the modify_user column
// row value set to "evirn"
```

Table *user_status_history*:

user_status_id	user_id	start_date	end_date
1	1	March 1, 2007	March 2, 2007
2	1	March 2, 2007	March 10, 2007
1	2	March 7, 2007	
3	1	March 10, 2007	

5. Future Enhancements

None