Time Tracker User version 3.2 Component Specification

1.Design

The Time Tracker User custom component is part of the Time Tracker application. It provides an abstraction of users. This component handles the persistence and other business logic required by the application.

The design is separated into 2 layers of management: The topmost layer are the Manager classes, which provide the functionality needed to update, retrieve and search from the data store. Batch operations are also available in the Manager classes. The next layer are the DAOs, which interact directly with the data store. There is also a sublayer, called the FilterFactory layer, which are responsible for building search filters which can be used to search the filter.

An additional layer is built on top of the first two layers. This layer is the J2EE layer, which consists of a Local, LocalHome, Delegate, and SessionBean classes for each of the different entities. The Local and LocalHome interfaces define the methods needed by the J2EE container to create the implementing classes. The Delegate, Local and SessionBean classes all implement/extend from their respective Manager interface, and therefore have the same set of business methods. The Delegate looks up the Local interface and routes all method calls to the Local interface. The Local interface then routes those calls to the SessionBean. The SessionBean finally routes those calls to the Manager implementation, where the actual business logic resides.

The final version of 3.1 included most of the functionality for the J2EE layer. The design makes some slight modifications and improves upon the documentation provided by this final version:

- No more state is maintained within the SessionBeans. The SessionBean 'looks up' the relevant Manager using a ManagerFactory. The UserManagerFactory acts like a singleton, and functions as a global access point for the Manager.
- A sample Deployment Descriptor is provided in the docs directory.
- A new Sequence Diagram was added to depict the J2EE functionality.
- A note was added to the Use Cases, saying that the application may now run under a J2EE container. Since the requirements for the new changes are not actual business functionality, but rather more on the inner functionality of the component, it did not seem appropriate to introduce a new use case for the new requirements.
- The JBoss Transaction DataSource may be configured using ConnectionFactory.
- The design attempts to adhere to the J2EE specification of not allowing File access by delegating all file access to an external UserManagerFactory class. This is a pragmatic approach which has been proven to work in previous designs (see the Orpheus Application components, like Game Persistence, Administration Persistence components). File access does not occur within the SessionBean itself, but rather within external classes when the application is called. It is also possible for File Access to occur before any SessionBean calls occur, since the ConfigManager may be initialized beforehand.
- Note that the only other alternative to not using ConfigManager and configuration files will be to restrict usage of ANY TC components that utilize configuration files - these include the Connection Factory, Search Builder and ID Generator components, which are a core part of the given component.

The following issues have also been identified:

- The SecurityRole and Principal objects from Authorization component need to be made Serializable. The issue here is similar to the Serializable issue of the Filter classes.

1.1Design Patterns

The Strategy Pattern. *The DAO class such as UserDAO is a strategy for persisting and retrieving information from a data store.*

The Facade Pattern. *The UserManager class encapsulates some subsystem DAOs and other components to provide a unified API that makes it easier to manage the Users.*

The Factory Pattern. *The Factory pattern is used in the FilterFactory layer. The UserFilterFactory employ this pattern.*

The Business Delegate Pattern. *The Business Delegate pattern is utilized in the UserManagerDelegate class.*

The Sevice Locator Pattern. *The Service Locator pattern is utilized in the same classes as the Business Delegate pattern. These classes perform a dual-function as a business delegate and service locator.*

1.2Industry Standards

None

1.3Required Algorithms

There were no algorithms required and the component is straightforward enough. We will discuss the database schema, and the method of searching in this section.

1.3.1 Data Mapping

This section will deal with mapping the different values in the beans to their respective columns. The developer is responsible for generating the necessary SQL to insert/retrieve the data in the beans from the appropriate columns. The SQL will require some simple table joins at the most. Please refer to the ERD diagrams for more information.

1.3.1.1 User Class and user_account Table [see TimeTrackerUser_ERD.jpg]

- Column user_account_id maps to the id property in the TimeTrackerBean superclass of the User class.
  Note: The id generator is used to create a new user_account_id when a new record is inserted.
- Column company_id maps to the companyId property in the User class.
- Column account_status_id maps to the id of the Status property in the User class.
- Column user_name maps to the username property in the User class.
- Column password maps to the passowrd property in the User class.
- Column creation_date maps to the creationDate property in the TimeTrackerBean superclass of the User class.
- Column creation_user maps to the creationUser property in the

TimeTrackerBean superclass of the User class.
- Column modification_date maps to the modificationDate property in the TimeTrackerBean superclass of the User class.
- Column modification_user maps to the modificationUser property in the TimeTrackerBean superclass of the User class.

1.3.1.2 User Class and Contact [we discuss the relevant columns for the Search Filters; see TimeTrackerContact_ERD.jpg]

- Table contact_relation is used to join the contact table with the user. The join is performed on entity_id -> user_account_id and contact_type_id -> contact type for a "USER" type entity.

  The actual contact data is located within the contact table:

- Column first_name maps to the User's first name [used in createFirstNameFilter]
- Column last_name maps to the User's last name [used in createLastNameFilter]
- Column phone maps to the User's phone no. [used in createPhoneNumberFilter]
- Column email maps to the User's email address [used in createEmailFilter]

1.3.1.3 User Class and Address [we discuss the relevant columns for the Search Filters; see TimeTrackerContact_ERD.jpg]

- Table address_relation is used to join the address table with the user. The join is performed on entity_id -> user_account_id and address_type_id -> contact type for a "USER" type entity.

  The actual address data is located within the address table:

- Column line1 maps to the User's Address [used in createAddressFilter]
- Column line2 maps to the User's Address [used in createAddressFilter]
- Column city maps to the User's city [used in createCityFilter]
- Column state_name_id will map to the User's State [used in createStateFilter]
- Column zip_code maps to the User's zipcode [used in createZipCodeFilter]

1.3.2 Searching

Searching is executed in this component by using the Search Builder component and the FilterFactory layer. First off, the developer needs to develop a search query off which to base the search on. The search query will retrieve all the necessary attributes, and join with the tables of any possible criterion. Once a query has been defined, in the constructor, the appropriate FilterFactory will be initialized with the column names used in the query. This FilterFactory can then be returned by the appropriate getFilterFactory method.

The user may then use the Factory to build the search filters. Once the filters are provided back to the DAO implementation, it may use the DatabaseSearchStrategy to build the search. The DatabaseSearchStrategy will then add the necessary WHERE clauses to constrain the search to the search criterion specified in the Filters.

Example:

The following query may be used as the context for searching the Users (we reduce the retrieved data to only the id, but developer may modify it to retrieve all relevant fields):

```
SELECT
      user_account_id
FROM
      user_account
INNER JOIN
      contact_relation
      ON
      contact_relation.entity_id = user_account_id
INNER JOIN
      contact
      ON
      contact.contact_id = contact_relation.contact_id
INNER JOIN
      address_relation
      ON
      address_relation.entity_id = user_account_id
INNER JOIN
      address
      ON
      address.address_id = address_relation.address_id
WHERE
```

From this context, the Search Builder can then add the different WHERE clauses, depending on the filter that was provide. For example, if a filter for the first name was created using createFirstNameFilter, then the Search Builder would add:

```
WHERE (continued from above)
      contact.first_name = [value of filter]
```

To accomplish this, in the constructor of the DAO, the DbUserFilterFactory must be configured according to the context query that was used. In this example, the mapped value for the FIRST_NAME_COLUMN_NAME should be "contact.first_name".

1.3.3 Auditing

The method implementation notes contain the audit header information that should be used when performing the audit. The following clarifies how to create AuditDetail objects to add to the AuditHeader. When creating a new entry (audit header action type is CREATE), then the oldValue for each detail is null. When deleting an entry (audit header action type is DELETE), then the newValue for each detail is null. When updating an entry, the old value is retrieved from the datastore to populate the AuditDetail's old value.

1.4Component Class Overview

Package com.topcoder.time.tracker.user

UserManager (interface)

This interface represents the API that may be used in order to manipulate the various details involving a Time Tracker User. CRUDE and search methods are provided to manage the Users inside a persistent store. There are also methods that for the manipulation of various roles for the Time Tracker user (The actual authorization of a User's actions can be done with the Authorization component).

UserDAO (interface)

This is an interface definition for the DAO that is responsible for handling the retrieval, storage, and searching of Time Tracker User data from a persistent store. It is also responsible for generating ids for any entities within it's scope, whenever an id is required.

UserFilterFactory (interface)

This interface defines a factory that is capable of creating search filters used for searching through Time Tracker Users. It offers a convenient way of specifying search criteria to use. The factory is capable of producing filters that conform to a specific schema, and is associated with the UserManager that supports the given schema.

BaseFilterFactory (interface)

This is a base FilterFactory interface that provides filter creation methods that may be used for filters of any Time Tracker Bean. It encapsulates filters for the common functionality - namely, the creation and modification date and user.

User

This is the main data class of the component, and includes getters and setters to access the various properties of a Time Tracker User, This class encapsulates the user's account information within the Time Tracker component. It contains the user's login and authentication details, contact information, and account status.

Status

This is an enumeration that is used to distinguish between the different types of Status that may be assigned to a user. At the moment, ACTIVE, INACTIVE and LOCKED status may be assigned.

UserManagerImpl implements UserManager

This is a default implementation of the UserManager interface. it utilizes instances of the UserDAO in order to fulfill the necessary CRUDE and search operations defined for the Time Tracker User.

UserManagerFactory

This class is used to create UserManager implementations to use by the delegate classes.

Package com.topcoder.time.tracker.user.db

DbUserDAO implements UserDAO

This is an implementation of the UserDAO interface that utilizes a database with the schema provided in the Requirements Section of Time Tracker 3.1. It delegates certain Contact-related information to the Time Tracker Contact component.

DbUserFilterFactory

This is an implementation of the UserFilterFactory that may be used for building searches in the database.

1.5Component Exception Definitions

DataAccessException

This exception is thrown when a problem occurs while this component is interacting with the persistent store. It is thrown by all the DAO and Utility interfaces (and their respective implementations).

UnrecognizedEntityException

This exception is thrown when interacting with the data store and an entity cannot be recognized. It may be thrown when an entity with a specified identifier cannot be found. It is thrown by all the Utility and DAO interfaces (and their implementations).

ConfigurationException

This exception is thrown when there is a problem that occurs when configuring this component.

BatchOperationException

This exception is thrown when there is a problem that occurs during batch operations.

.

1.6Thread Safety

This component is not completely thread-safe, The Bean instances are not thread safe, and it is expected to be used concurrently only for read-only access. Otherwise, each thread is expected to work with its own instance.

The UserManagerImpl class is required to be thread safe, and achieves this via the thread safety of the implementations of the DAO Layer, and the FilterFactory Layer. AuthorizationPersistence is not thread-safe, so synchronization/synchronized blocks may be needed in the critical areas.

The DAOLayer is made thread safe through the use of transactions, and is achieved with the transaction level of READ_COMMITTED.

Thread-safety of the J2EE layer is dependent on the statelessness and thread safety of the manager classes, and also on the application container controlling them. The only thing to pay attention to is that the inner state of the classes are not modified after creation.

2.Environment Requirements

2.1Environment

Java 1.4

2.2TopCoder Software Components

DB Connection Factory 1.0 – for creating the DB connections

Base Exception 1.0 – base class for custom exception is taken from it

Object Factory 2.0 – is indirectly used to configure the component.

ID Generator 3.0 – for generating IDs in the persistence implementation.

Search Builder 1.3.1 – is used to perform the searches

Authentication Factory 2.0 - is used to perform user authentication.

Authorization 2.1 - is used to associate roles with users.

TypeSafe Enum 1.0 - is used to enumerate the different Status of the User.

ConfigManager 2.1.5 – is used configure certain portions of the component (also used by Object Factory).

Time Tracker Audit 3.2 – is used to perform the optional audit.

Time Tracker Contact 3.2 – is used to persist and retrieve User Contact information and Address

Time Tracker Common 3.2 – is used to provide the TimeTrackerBean base class.

JNDI Context Utility 1.0 - for retrieving the LocalHome interface implementations from the container.

2.3Third Party Components

None

3.Installation and Configuration

3.1Package Names

com.topcoder.time.tracker.user
com.topcoder.time.tracker.user.db
com.topcoder.time.tracker.user.filterfactory
com.topcoder.time.tracker.user.j2ee

3.2Configuration Parameters

This component relies on Object Factory 2.0 for configuration that is based from a file.

For UserAuthenticator:

The UserAuthenticator utilizes the following properties for configuration (note that it inherits the configuration properties of the parent class AbstractAuthenticator if Caching or PrincipalKey configuration is desired):

objectFactoryNamespace - This is the namespace that is used to create a ConfigManagerSpecification factory that will be used to initialize the ObjectFactory used to create the UserDAO implementation to use. **Required.**

userDao - This property will contain subproperties that define how to initialize the userDao that will be used. **Required.**

userDao.classname - The classname of the UserDAO implementation to use. It will be provided as a method parameter to the createObject method of Object Factory. **Required.**

userDao.identifier - An optional identifier that may also be provided to the createObject method of ObjectFactory. **Optional.**

For UserManagerDelegate:

Property Name: *contextName*
Property Description: This is the context name used to retrieve the home object of the respective session bean. If not specified, then the default context provided by JNDIUtils

is used.
Is Required: *Optional*

3.3Dependencies Configuration

All the dependencies are to be configured according to their component specifications.

Do note that the default configuration for Search Builder is desired to be able to build the search string correctly.

4.Usage Notes

4.1Required steps to test the component

Extract the component distribution.

Follow Dependencies Configuration.

Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2Required steps to use the component

Configure the dependency components.

4.3 Demo

Please see the ejb-jar.xml file in docs directory for the deployment descriptor for the J2EE layer of this component.

We will assume here that everything is configured properly. Try/catch clauses have been removed to enhance clarity.

```
// Create a UserManager using the DB DAOs, AuthPersistence and User Authenticator.
UserManager userManager = new UserManager(userDao, authPersistence, "userAuthenticator");

//  4.3.1 Creating a new User
User newUser = new User();
newUser.setUsername("TCSDESIGNER");
newUser.setPassword("5yks809");
newUser.setCompanyId(topcoderId);
newUser.setContact(newUserContact);
newUser.setAddress(newUserAddress);
newUser.setStatus(Status.ACTIVE);
newUser.setCreationUser("TCSADMIN");
newUser.setModificationUser("TCSADMIN");

// Register the user with the manager, with auditing.
userManager.createUser(newUser, true);

// 4.3.2 Changing the User Details

// Retrieve a user from the manager
User deactivatingUser = userManager.getUser(accountIdToDeactivate);

// Update the user's details.
deactivatingUser.setStatus(Status.INACTIVE);
```

```
deactivatingUser.setModificationUser("AUTOMATED_EXPIRY_TOOL");

// Update the user in the manager
userManager.updateUser(deactivatingUser, true);

// 4.3.3 Search for a group of active users belonging to ACME company and delete them.
The delete operation should be atomic.

// Define the search criteria.
UserFilterFactory filterFactory = userManager.getUserFilterFactory();
List criteria = new ArrayList();
criteria.add(filterFactory.createCompanyIdFilter(acmeCompanyId));
criteria.add(filterFactory.createStatusFilter(Status.ACTIVE));

// Create a search filter that aggregates the criteria.
Filter searchFilter = new AndFilter(criteria);

// Perform the actual search.
User[] matchingUsers = userManager.searchUsers(searchFilter);

// Delete the users using atomic mode; auditing is performed.
userManager.removeUsers(matchingUsers, true);

// 4.3.3 Manipulating User Authorization Roles

// Make the new User into a SuperUser by giving it the SuperUser role; We assume
AuthorizationPersistence has been initialized.

SecurityRole role = (SecurityRole)authPersistence.searchRoles("SuperUser").iterator().next();
userManager.addRoleForUser(newUser, role);

// Remove all the roles from the deactivatingUser
userManager.clearRolesFromUser(newUser);

// 4.3.4 User Authentication

// Authenticate a username/password combination.
userManager.authenticateUser('TCSDESIGNER', 'TCSPASSWORD');

// 4.3.5 Creating a Delegate

UserManager userManagerDelegate = new UserManagerDelegate(“applicationNamespace”);
// Operations may then be performed on the delegate similar to those outlined in the previous
// sections of the demo.
```

5.Future Enhancements

Provide implementations for different RDBMS.