Resource Management 1.1 Component Specification

Aspects of the design that are changed in version 1.1 are blue.

Additions in version 1.1 are red.

1. Design

The Resource Management component provides resource management functionalities. A resource can be associated with a project, phase and submission. Each resource will have a role which identifies the resource's responsibilities for the associated scope. A set of resources can be created, updated or searched for a project. Notifications can also be assigned and unassigned to users. The persistence logic for resources and related entities is pluggable.

For development, this component is split into two parts. The main part will need to be done first, as the persistence part can not be done without the object model classes (in the main part) are complete.

- Main part: This development project will be responsible for all classes and interfaces in the com.topcoder.management.resource package and the com.topcoder.management.resource.search classes. The ResourcePersistenceException will also fall in this development project because it is declared to be thrown in the ResourceManager interface.
- Persistence part: This development project will be responsible for the classes in the com.topcoder.management.resource.persistence and com.topcoder.management.resource.persistence.sql packages. Development of the ResourcePersistenceException will not fall in this development project, as it will already have been done in the main part.

Version 1.1 provides two DAO implementations, one that manages its own transactions and is fully backward compatible, and one that relies on externally managed transactions. Both DAO implementations are identical in all respects other than transaction management.

This design realizes the requirements by refactoring most of the logic into an abstract superclass of both concrete DAOs. The template method pattern is used, with the abstract class performing the bulk of the logic for each method, but relying on the subclasses to open and close database connections. If transaction management is needed, it is incorporated into the implementations of these methods. Because of this design, none of the main logic of the component is duplicated, and the logic to open and close connections and the logic to commit transactions is only implemented a single time for each concrete class.

In addition to changing the transaction management approach, all of the parameters and return types for a persistence class used in a distributed transaction application like EJB need to be serializable. In order for this solution

to work, the Search Builder Filter class will need to be made serializable; this will be taken care of outside of this component.

1.1 Design Patterns

The ResourcePersistence interface and implementations uses the **Strategy Pattern**, as does the ResourceManager interface and implementations. The Delegate pattern is used to delegate filter building tasks to the Search Builder component.

The AbstractResourcePersistence class uses the **Template Method Pattern** by providing protected abstract openConnection(), closeConnection(), and closeConnectionOnError() methods that are implemented by subclasses and used for connection and potentially transaction management in all of the public methods of the class.

1.2 Industry Standards

SQL

1.3 Required Algorithms

The only complicated part of this component is the SQL queries needed in the SqlResourcePersistence class. Beyond this, nothing more complicated than a simple for loop or an if/else is needed in this component.

1.3.1 Sql Queries for AbstractResourcePersistence

This section lists the SQL queries and statements that will be needed by the SqlResourcePersistence class, accompanied by a written explanation of what to do, when more logic than just a sequence of SQL statements is needed. All of these SQL statements should be executed using PreparedStatements. A typical method of this class would look like (this example uses the "Update Notification Type" statement):

SQL statement to execute:

```
UPDATE notification_type_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE notification_type_id = ?
```

Sample Code:

```
// Open connection
Connection connection = openConnection();
// Create a prepared statement
PreparedStatement ps =
      connection.prepareStatement(< above text >);
// Use data in NotificationType passed to method to set
// parameters in prepared statement
ps.setString(1, notificationType.getName());
ps.setString(2, notificationType.getDescription());
ps.setString(3, notificationType.getModificationUser());
ps.setDate(4, notificationType.getModificationDate());
```

```
ps.setLong(5, notificationType.getId());
// execute PreparedStatement
ps.execute();
// close connection
closeConnection(connection);
```

1.3.1.1 Update Resource

```
First, update the resource table
UPDATE resource
SET resource_role_id = ?, project_id = ?, phase_id = ?,
modify_user = ?, modify_date = ?
WHERE resource_id = ?
```

Next, check if there is a submission entry for the resource:

```
SELECT submission_id
FROM resource_submission
WHERE resource_id = ?
```

What to do with the submission depends on whether the previous query returns a row (there is a previous submission entry), and whether the Resource passed to the method has a current submission entry (its getSubmission() method returns a non-null value).

- No previous submission and no current submission:
  - Do nothing
- No previous submission and has current submission:
  - Add a row to resource_submission. Use the second SQL statement in the Add Resource section)
- Has previous submission and no current submission:
  - Remove a row from resource_submission. Use the second SQL statement in the Delete Resource section
- Has previous submission and has current submission:
  - Update a row in resource_submission table using the SQL statement:

    ```
    UPDATE resource_submission
    SET submission_id = ?, modify_user = ?,
    modify_date = ?
    WHERE resource_id = ?
    ```

Look up all the existing properties for this resource (see the second query in the Load Resource section).

The task of updating the extended properties largely parallels the submission update. For each extended property name that appears either in the map of the Resource passed to this method, or in the existing properties in the database:

- No previous entry for property name and has current entry for property name:
  - Add a row to resource_info using the method given in the Add Resource section. Do not add the property if an entry for the name is not found in the resource_info_type_lu table.

- Has previous entry for property name and has current entry for property name
  - Look up resource_info_type_id using the third SQL statement in the Add Resource section.
  - Update a row in resource_info using the SQL statement
    ```
    UPDATE resource_info
    SET value = ?
    WHERE resource_id = ? AND resource_info_type_id =
    ?
    ```
- Previous entry for property name and no current entry for property name:
  - Look up the resource_info_type_id using the third SQL statement in the Add Resource section.
  - Remove the row in the resource_info table using the SQL statement
    ```
    DELETE FROM resource_info
    WHERE resource_id = ? and resource_info_type_id =
    ?
    ```

1.3.1.2 Delete Resource

First, delete all extended properties for the resource:

```
DELETE FROM resource_info
WHERE resource_id = ?
```

Then delete entry from resource_submission table:

```
DELETE FROM resource_submission
WHERE resource_id = ?
```

Finally delete entry in resource table

```
DELETE FROM resource
WHERE resource_id = ?
```

1.3.1.3 Add Resource

```
INSERT INTO resource
(resource_id, resource_role_id, project_id, phase_id,
submission_id, create_user, create_date, modify_user,
modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

If submission is not null, then insert submission entry via:

```
INSERT INTO resource_submission
(resource_id, submission_id, create_user, create_date,
modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?)
```

For each {name, value} extended property pair in the extended properties map, first look up the resource_info_type_id for the name using:

```
SELECT resource_info_type_id
FROM resource_info_type_lu
WHERE name = ?
```

If there is an entry for the name (i.e. the previous query returned a row), then check if there is already add an info entry for the resource using the following query. If there was no entry for the name, the extended property is silently ignored and will not be persisted.

```
INSERT INTO resource_info
(resource_id, resource_info_type_id, value, create_user,
create_date, modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

The value to insert is the .toString() return of the value in the properties map.

1.3.1.4 Load Resource

```
SELECT resource_id, resource_role_id, project_id, phase_id,
submission_id, create_user, create_date, modify_user, modify_date
FROM resource LEFT OUTER JOIN resource_submission
ON resource.resource_id = resource_submission.resource_id
WHERE resource_id = ?
```

To select all external properties:

```
SELECT resource_info_type_lu.name, resource_info.value
FROM resource_info INNER JOIN resource_info_type_lu ON
(resource_info.resource_info_type_id =
resource_info_type_lu.resource_info_type_id)
WHERE resource_id = ?
```

1.3.1.5 Add Notification

```
INSERT INTO notification
(project_id, external_ref_id, notification_type_id, create_user,
create_date, modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

Note that for this table, both the create_user and modify_user are set to the operator passed to the method, as are the modify_user and modify_date. The other add and update methods in this class use the values in the object passed to the methods.

1.3.1.6 Remove Notification

```
DELETE FROM notification
WHERE project = ? AND external_ref_id = ? AND
notification_type_id = ?
```

1.3.1.7 Add Notification Type

```
INSERT INTO notification_type_lu
(notification_type_id, name, description, create_user,
create_date, modify_user, modify_date)
VALUES(?, ?, ?, ?, ?, ?, ?)
```

1.3.1.8 Delete Notification Type

```
DELETE FROM notification_type_lu
WHERE notification_type_id = ?
```

1.3.1.9 Update Notification Type

```
UPDATE notification_type_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE notification_type_id = ?
```

1.3.1.10 Load Notification Type

```
SELECT notification_type_id, name, description, create_user,
create_date, modify_user, modify_date
FROM notification_type_lu
WHERE notification_type_id = ?
```

1.3.1.10.1 Load Notification Types

This is the same as above, but the WHERE clause becomes

WHERE `notification_type_id` IN (<id_values>)

1.3.1.11 Add Resource Role

```
INSERT INTO resource_role_lu
(resource_role_id, name, description, phase_type_id, create_user,
create_date, modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

1.3.1.12 Delete Resource Role

```
DELETE FROM resource_role_lu
WHERE resource_role_id = ?
```

1.3.1.13 Update Resource Role

```
UPDATE resource_role_lu
SET phase_type_id = ?, name = ?, description = ?, phase_type_id =
?, modify_user = ?, modify_date = ?
WHERE resource_role_id = ?
```

1.3.1.14 Load Resource Role

```
SELECT resource_role_id, phase_type_id, name, description,
create_user, create_date, modify_user, modify_date
FROM resource_role_lu
WHERE resource_role_id = ?
```

1.3.1.14.1 Load Resource Roles

This is the same as above, but the WHERE clause becomes

WHERE resource_role_id IN (<id_values)

1.3.1.15 Load notification

```
SELECT project_id, external_ref_id, notification_type_id,
create_user, create_date, modify_user, modify_date
FROM notification
WHERE project_id = ? AND external_ref_id = ? AND
notification_type_id = ?
```

For getting multiple notifications at the same time, the WHERE clause becomes:

```
WHERE (project_id = ? AND external_ref_id = ? AND
notification_type_id = ?) OR (project_id = ? AND external_ref_id
= ? AND notification_type_id = ?) OR …
```

1.3.1.16 Load resources

The loadResources method is designed to retrieve all resources with matching ids with the use of only two database queries. This keeps the number of expensive database queries down and will improve the performance of the application.

```
SELECT resource.resource_id, resource_role_id, project_id,
phase_id, submission_id, resource.create_user,
resource.create_date, resource.modify_user, resource.modify_date
FROM resource LEFT OUTER JOIN resource_submission ON
resource.resource_id = resource_submission.resource_id WHERE
resource.resource_id IN (<id list>);
```

To select all external properties:

```
SELECT resource_info.resource_id, resource_info_type_lu.name,
resource_info.value  FROM resource_info INNER JOIN
resource_info_type_lu ON (resource_info.resource_info_type_id =
resource_info_type_lu.resource_info_type_id) WHERE
resource_info.resource_id IN (<id list>);
```

The selected external properties should then be matched up with the resources.

1.3.2 Transaction management

Transactions are managed in the concrete subclasses of AbstractResourcePersistence, which are SqlResourcePersistence and UnmanagedTransactionResourcePersistence. SqlResourcePersistence handles its own transactions while UnmanagedTransactionResourcePersistence relies on externally managed transactions.

Connections are opened using the openConnection() method. When this method is called on SqlResourcePersistence, a connection will be opened using the connectionFactory and (if not null) connectionName. The retrieved Connection will be opened, and setAutoCommit(false) will be called on it before returning. UnmanagedTransactionResourcePersistence will follow the same logic, except that setAutoCommit() won't be called.

In the closeConnection() method, SqlResourcePersistence will commit the transaction on the Connection before closing it, by calling the commit() method

on the Connection. UnmanagedTransactionResourcePersistence will simply call close() on the given Connection without first committing anything.

If an error occurs while preparing or executing PreparedStatements or processing ResultSets, the connection will need to be disposed of before the ResourcePersistenceException is thrown. If there is any transaction, it should be rolled back rather than committed, so a separate method is provided for this case - closeConnectionOnError(). The implementation of this method for UnmanagedTransactionResourcePersistence should be identical to that class's implementation of closeConnection(). SqlResourcePersistence will need to call rollback() instead of commit() on the Connection.

The code needs to ensure that either closeConnection() or closeConnectionOnError() is called, regardless of where exceptions are thrown. In SqlResourceManagement, the actual connection should be closed in a finally clause, so that the connection is always closed.

1.4 Component Class Overview

ResourceManager:

The ResourceManager interface provides the ability to persist, retrieve and search for persisted resource modeling objects. This interface provides a higher level of interaction than the ResourcePersistence interface. For example, the updateResource method will determine if the resource is new, and if so, assign it an id before persisting it, whereas the ResourcePersistence interface will simply fail in this situation. The methods in this interface break down into dealing with the 4 main modeling classes in this component, and the methods for each modeling class are quite similar.

Implementations of this interface are not required to be thread safe.

AuditableResourceStructure:

The AuditableResourceStructure is the base class for the modeling classes in this component. It holds the information about when the structure was created and updated. This class simply holds the four data fields needed for this auditing information and exposes both getters and setters for these fields.

All the methods in this class do is get/set the underlying fields, so this class will be very easy to develop.

This class is highly mutable. All fields can be changed.

Resource:

The Resource class is the main modeling class in this component. It represents any arbitrary resource. The Resource class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

ResourceRole:

The ResourceRole class is the second modeling class in this component. It represents a type of resource and is used to tag instances of the Resource class as playing a certain role. The ResourceRole class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

NotificationType:

The NotificationType class is the third modeling class in this component. It represents a type of notification. It is orthogonal to the Resource and ResourceRole classes. This class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

Notification:

The Notification class is the final modeling class in this component. It represents a notification, which is an association between an external id (the use and meaning of this field is up to the user of the component), a project, and a notification type. This class is simply a container for these data fields. All data fields (directly in this class) are immutable and have only getters.

This class is mutable because its base class is mutable.

ResourceFilterBuilder:

The ResourceFilterBuilder class supports building filters for searching for Resources. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

ResourceRoleFilterBuilder:

The ResourceRoleFilterBuilder class supports building filters for searching for ResourceRoles. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

NotificationFilterBuilder:

The NotificationFilterBuilder class supports building filters for searching for Notifications. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

NotificationTypeFilterBuilder:

The NotificationTypeFilterBuilder class supports building filters for searching for NotificationTypes. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

ResourcePersistence:

The ResourcePersistence interface defines the methods for persisting and retrieving the object model in this component. This interface handles the persistence of the four classes that make up the object model – Resources, ResourceRoles, Notifications, and NotificationTypes. This interface is not responsible for searching the persistence for the various entities. This is instead handled by a ResourceManager implementation.

Implementations of this interface are not required to be thread-safe or immutable.

AbstractResourcePersistence:

The AbstractResourcePersistence class implements the ResourcePersistence interface, in order to persist to the database structure in the resource_management.sql script. It contains most of the logic that was in the SqlResourcePersistence class in version 1.0.1.

This class does not cache a Connection to the database. Instead, it uses the concrete implementation of the openConnection() method of whatever subclass is in use to acquire and open the Connection. After the queries are executed and the result sets processed, it uses the closeConnection() method to dispose of the connection. If the operation fails, closeConnectionOnError() is called instead. This allows the transaction handling logic to be implemented in subclasses while the Statements, queries, and ResultSets are handled in the abstract class.

Most methods in this class will just create and execute a single PreparedStatement. However, some of the Resource related methods need to execute several PreparedStatments in order to accomplish the update/insertion/deletion of the resource.

This class is immutable and thread-safe in the sense that multiple threads can not corrupt its internal data structures. However, the results if used from multiple threads can be unpredictable as the database is changed from different threads. This can equally well occur when the component is used on multiple machines or multiple instances are used, so this is not a thread-safety concern.

SqlResourcePersistence:

The SqlResourcePersistence class in version 1.1 is completely compatible with the SqlResourcePersistence class in version 1.0.1, and has no additional functionality. However, the bulk of the logic that was in this class in version 1.0.1 has been moved into an abstract superclass that is the base for UnmanagedTransactionResourcePersistence as well. The only logic remaining in this class is that of opening connections and managing transactions, and the only methods implemented in this class are openConnection(), closeConnection(), and closeConnectionOnError(), which are concrete implementations of the corresponding protected abstract methods in AbstractResourcePersistence and are used in the context of a Template Method pattern.

This class is immutable and thread-safe in the sense that multiple threads can not corrupt its internal data structures. However, the results if used from multiple threads can be unpredictable as the database is changed from different threads. This can equally well occur when the component is used on multiple machines or multiple instances are used, so this is not a thread-safety concern.

UnmanagedTransactionResourcePersistence:

The UnmanagedTransactionResourcePersistence class is a new class in version 1.1. It performs exactly the same tasks as SqlResourcePersistence, but is designed to be used with externally managed transactions. The implementations of openConnection(), closeConnection(), and closeConnectionOnError() in this class do not call commit(), setAutoCommit(), or rollback(), as the transaction is expected to be handled externally to the component.

This class is immutable and thread-safe in the sense that multiple threads can not corrupt its internal data structures. However, the results if used from multiple threads can be unpredictable as the database is changed from different threads. This can equally well occur when the component is used on multiple machines or multiple instances are used, so this is not a thread-safety concern.

PersistenceResourceManager:

The PersistenceResourceManager class implements the ResourceManager interface. It ties together a persistence mechanism, several Search Builder searching instances (for searching for various types of data), and several id generators (for generating ids for the various types). This class consists of several methods styles. The first method style just calls directly to a corresponding method of the persistence. The second method style first assigns values to some data fields of the object before calling a persistence method. The third type of method uses a SearchBundle to execute a search and then uses the persistence to load an object for each of the ids found from the search.

This class is immutable and hence thread-safe.

1.5 Component Exception Definitions

ResourcePersistenceException:

The ResourcePersistenceException indicates that there was an error accessing or updating a persisted resource store. This exception is used to wrap the internal error that occurs when accessing the persistence store. For example, in the SqlResourcePersistence implementation it is used to wrap SqlExceptions.

This exception is initially thrown in ResourcePersistence implementations and from there passes through ResourceManager implementations and back to the caller. It is also thrown directly by some ResourceManager implementations.

IdAlreadySetException:

The IdAlreadySetException is used to signal that the id of one of the resource modeling classes has already been set. This is used to prevent the id being changed once it has been set.

This exception is initially thrown in the 3 setId methods of the resource modeling classes.

1.6 Thread Safety

This component is not thread safe. This decision was made because the modeling classes in this component are mutable while the persistence classes make use of non-thread-safe components such as Search Builder. Combined with there being no business oriented requirement to make the component thread safe, making this component thread safe would only increase development work and needed testing while decreasing runtime performance (because synchronization would be needed for various methods). These tradeoffs can not be justified given that there is no current business need for this component to be thread-safe.

This does not mean that making this component thread-safe would be particularly hard. Making the modeling classes thread safe can be done by simply adding the synchronized keyword to the various set and get methods. The persistence and manager classes would be harder to make thread safe. In addition to making manager and persistence methods synchronized, there would need to be logic added to handle conditions that occur when multiple threads are manipulating the persistence. For example, “Resource removed between SearchBuilder query and loadResource call”.

Version 1.1 doesn't make any changes that impact thread safety. The component is still not safe for use from multiple threads. However, transactions can be spread over multiple classes and multiple threads if they are managed externally to the component.

2. Environment Requirements

2.1 Environment

Java 1.4+ is required for compilation, testing, or use of this component

2.2 TopCoder Software Components

- Search Builder 1.1 or greater: Used for searching for resources and related resources.
- DB Connection Factory 1.0: Used in SQL persistence implementation to connect to the database. Also used by the Search Builder component.
- Custom Result Set 1.1: Returned by the Search Builder component and used to retrieve the ids of the items selected by the search.
- ID Generator 3.0: Used to create ids when new Resources and related objects are created.
- Configuration Manager 2.1.4: Used to configure the DB Connection Factory and Search Builder components. Can also be used with the Object Factory component. Not used directly in this component.

- Object Factory 2.0: Can be used to create ResourceManager implementations, particularly the PersistenceResourceManager class. There is no compile time or runtime dependency on this component.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.resource
com.topcoder.management.resource.persistence
com.topcoder.management.resource.persistence.sql
com.topcoder.management.resource.search

3.2 Configuration Parameters

No direct configuration is used for this component. The Object Factory component can be used to create SqlResourcePersistences and PersistenceResourceManagers if desired. The SearchBundles and IDGenerators needed can be configured or created programmatically. For configuration of these objects, see the relevant component specifications.

When using the SQL database structure given in the resource_management.sql script, the SearchBundles passed to the PersistenceResourceManager should be configured to use the following contexts (queries minus where clause):

For searching Resources: (Note that this query allows only a single extension property name/value pair to be queried, combined with any other filters. It will not allow queries of multiple extension property names/values: the query will not fail, but it will never return any rows.)

```
SELECT resource_id
FROM resource
LEFT OUTER JOIN resource_submission
      ON resource.resource_id = resource_submission.resource_id
LEFT OUTER JOIN resource_info
      ON resource.resource_id = resource_info.resource_id
LEFT OUTER JOIN resource_info_type_lu
      ON resource_info.resource_info_type_id =
            resource_info_type_lu.resource_info_type_id
WHERE
```

For searching ResourceRoles

```
SELECT resource_role_id
FROM resource_role_lu
WHERE
```

For searching NotificationTypes

```
SELECT notification_type_id
FROM notification_type_lu
```

```
WHERE
```

For searching Notifications

```
SELECT external_ref_id, project_id, notification_type_id
FROM notification
WHERE
```

3.3 Dependencies Configuration

None

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

Install and configure the other TopCoder component following their instructions. Then follow section 4.1 and the demo.

Demo:

4.2.1 Create a Resource and ResourceRole

```
ResourceRole resourceRole = new ResourceRole();
resourceRole.setName("Some Resource Role");
resourceRole.setDescription("This role plays some purpose");

// Note that it is not necessary to set any other field of the
// resource because they are all optional
Resource resource = new Resource();
resource.setResourceRole(resourceRole);

// The creation of notification types is entirely similar
// to the calls above.
```

4.2.2 Create persistence and manager

```
SqlResourcePersistence persistence =
      new SqlResourcePersistence(
            <connection factory loaded from configuration>)

ResourceManager manager = new PersistenceResourceManager(
      persistence,
      <search builders loaded from configuration: See Search
            Builder component for configuration details>,
      <id generators loaded from configuration: See ID
            Generator component for configuration details>);
```

4.2.3 *Save the created Resource and ResourceRole to the persistence*

```
// Note that this will assign an id to the resource and
// resource role
manager.updateResourceRole(resourceRole, “Operator #1”);
manager.updateResource(resource, “Operator #1”);
// The updating of notification types is entirely similar
// to the calls above

// The data can then be changed and the changes
// persisted
resource.setName(“Changed name”);
manager.updateResource(resource, “Operator #1”);
```

4.2.4 *Update All Resources For a Project*

```
long projectId = 1205;
// Removes any resources for the project not in the array
// and updates/adds those in the array to the persistence
manager.updateResources(new Resource[] {resource},
      project, “Operator #1”);
```

4.2.5 *Retrieve and search resources*

```
// Get a resource for a given id
Resource resource2 = manager.getResource(14402);
// The properties of the resource can then be queried
// and used by the client of this component

// Search for resources
// Build the filters – this example shows searching for
// all resources related to a given project and of a
// given type and having an extension property of a given name
Filter projectFilter =
      ResourceFilterBuilder.createProjectIdFilter(953);
Filter resourceTypeFilter =
      ResourceFilterBuilder.createResourceRoleIdFilter(1223);
Filter extensionNameFilter =
      ResourceFilterBuilder.createExtensionPropertyNameFilter(
            “Extension Prop Name”);
Filter fullFilter =
      SearchBundle.createAndFilter(SearchBundle.createAndFilter(
            projectFilter, resourceTypeFilter),
            extensionNameFilter);

// Search for the Resources
Resource[] matchingResources =
      manager.searchResources(fullFilter);

// ResourceRoles, NotificationTypes, and Notifications can be
// searched similarly by using the other FilterBuilder classes
// and the corresponding ResourceManager methods.  They can
// also be retrieved through the getAll methods
ResourceRole[] allResourceRoles = manager.getAllResourceRoles();
NotificationType[] allNotificationTypes =
      manager.getAllNotificationTypes();
```

4.2.6 *Add/Remove notifications*

```
// Note that it is up to the application to decide what the
// user/external ids represent in their system
manager.addNotifications(new long[] {1, 2, 3}, 953, 192,
      “Operator #1”);
manager.removeNotifications(new long[] {1, 2, 3}, 953, 192,
      “Operator #1”);
```

4.2.7 *Retrieve notifications*

```
long[] users = manager.getNotifications(953, 192);
// This might, for example, represent the users to which an email
// needs to be sent.  The client could then lookup the user
// information for each id and send the email.

// Searches for notifications can also be made using an API
// that precisely parallels that shown for Resources in 4.3.5.
// When searching is used, full-fledges Notification instances
// are returned.

SqlResourcePersistence demo part:
ResourceOperationsDemo :
// Clear resource related tables.
        DBTestUtil.clearTables(new String[]{"resource_info",
"resource_info_type_lu",
            "resource_submission", "resource",
"resource_role_lu"});
        // Create a role in advance.
        ResourceRole role = DBTestUtil.createResourceRole(5);
        persistence.addResourceRole(role);

        long resourceId = 1;

        // create a resource instance with id equals 1, project
id 1 and phase id 1.
        Resource resource = DBTestUtil.createResource(resourceId,
1, 1);

        // Create the resource instance in the database.
        this.persistence.addResource(resource);

        // Load the resource with id equals 1 from database.
        Resource result =
this.persistence.loadResource(resourceId);
        assertEquals("id should be 1", 1, result.getId());
        assertEquals("project id should be 1", 1,
result.getProject().longValue());
        assertEquals("phase id should be 1", 1,
result.getPhase().longValue());

        resource.setPhase(new Long(2)); // Modified the phase
        resource.setProject(new Long(3)); // Modified the project.
        long submissionId = 121;
        resource.setSubmission(new Long(submissionId)); // set a
submission.

        resource.setModificationUser("me"); // set the users that
modified the record.
```

```
        resource.setModificationTimestamp(new Date()); // set the
timestamp of the modification

        // The resource has been modified, update the database.
        this.persistence.updateResource(resource);

        // A Resource can also have extended properties.
        resource.setProperty("name", "topcoder1");
        resource.setProperty("age", "25");

        // Save the properties of the resource to database.
        this.persistence.updateResource(resource);

        // Finally, if you want to delete the resource from db.
        this.persistence.deleteResource(resource);

LoadResourcesDemo:
       // Create a role in advance.
        ResourceRole role = DBTestUtil.createResourceRole(5);
        persistence.addResourceRole(role);

        // Create 3 resource instances with id 1, 2, 3.
        for (int i = 0; i < 3; i++) {
            long resourceId = i + 1;
            Resource resource =
DBTestUtil.createResource(resourceId, 1, 1);
            this.persistence.addResource(resource);
        }

        // Load the 3 resources from db by providing a list of
ids..
        Resource[] results = this.persistence.loadResources(new
long[]{1, 2, 3});
        assertEquals("3 resources are loaded from db", 3,
results.length);

        // no resource has id 4, 5 or 6, so it will still load 3
resources from db.
        results = this.persistence.loadResources(new long[]{1, 2,
3, 4, 5, 6});
        assertEquals("3 resources are loaded from db", 3,
results.length);

ResourceRoleOperationsDemo:

        // Create a new ResourceRole instance with id 999999.
        long resourceRoleId = 999999;
        ResourceRole role =
DBTestUtil.createResourceRole(resourceRoleId);

        // insert the ResourceRole to database.
        persistence.addResourceRole(role);

        // Load the resource role from db.
        ResourceRole result =
persistence.loadResourceRole(resourceRoleId);
```

```
        assertEquals("id should be 999999", resourceRoleId,
result.getId());

        // update the resource role.
        role.setPhaseType(new Long(2));
        role.setName("developer");
        role.setDescription("testing, testing");
        persistence.updateResourceRole(role);

        // Delete the resource role.
        persistence.deleteResourceRole(role);

        // Load a list of resource roles.
        ResourceRole[] roles = persistence.loadResourceRoles(new
long[]{1, 2, 3});
```

```
NotificationOperationsDemo:
     // clear notification related tables first.
        DBTestUtil.clearTables(new String[]{"notification",
"notification_type_lu"});

        // Create a notification type in advance.
        NotificationType type =
DBTestUtil.createNotificationType(2);
        persistence.addNotificationType(type);

        // Assume there is a Resource instance with id 1,
addNotification will add it to the notification list.
        long projectId = 2;
        long notificationTypeId = 2;
        persistence.addNotification(1, projectId,
notificationTypeId, "operator");

        // Now I want to delete user 1 from the notification list.
        persistence.removeNotification(1, projectId,
notificationTypeId, "operator");

        // Load a notification from the db to see if a user is in
the notification list.
        persistence.loadNotification(1, projectId,
notificationTypeId);

        // Load a list of notifications from db.
        persistence.loadNotifications(new long[]{1}, new
long[]{projectId}, new long[]{notificationTypeId});
```

```
NotificationTypeOperationsDemo:
        // Create a new NotificationType with id equals 2
        long notificationTypeId = 2;
        NotificationType type =
DBTestUtil.createNotificationType(notificationTypeId);

        // Insert the notification type in the db.
        persistence.addNotificationType(type);
```

```
        // Load a notification type instance from the db with its
id.
        NotificationType result =
persistence.loadNotificationType(notificationTypeId);
        assertEquals("id should be 2", notificationTypeId,
result.getId());

        // Updated the description of the notification type.
        type.setDescription("a new description for the
notification type");
        type.setName("a new name");

        type.setModificationUser("opeartor");
        type.setModificationTimestamp(new Date());
        // Update the notification type in the db.
        persistence.updateNotificationType(type);

        // Delete a notification type from db.
        persistence.deleteNotificationType(type);

        // Load  a list of notification types from the db.
        persistence.loadNotificationTypes(new long[]{1, 2, 3});
```

4.2.8 *Managed transaction demo*

Version 1.1 of this component enables the use of externally managed transactions. One scenario a customer might use is an EJB application using container managed transactions.

A user wishing to take advantage of this new functionality in an EJB application will need to implement a Session Bean and a Home and Local or Remote interface. In this demo, a possible implementation of a Session Bean along with a Local Home and Local interface is shown, and a possible deployment descriptor that will ensure that all methods are executed in the scope of a transaction.

The following session bean implements the ResourceManager interface, allowing a client access to all of the functionality provided by ResourceManager through a session bean with container managed transactions.

If a business operation fails, the setRollbackOnly() method is used to inform the EJB container that the current transaction should be rolled back at the appropriate time rather than committed.

ResourceBean.java:

```
package com.acme.resource;
import java.util.*;
import javax.ejb.*;
import com.topcoder.management.resource.*;
import com.topcoder.management.resource.persistence.*;
import com.topcoder.util.config.ConfigManager;

public class ResourceBean implements SessionBean, ResourceManager {

        SessionContext context;
        ResourceManager manager;

        // create the bean with a manager configured from the given namespace
        ejbCreate(String namespace) throws CreateException{
                try{
                SqlResourcePersistence persistence = new SqlResourcePersistence(
```

```
                        <connection factory loaded from configuration>)
                ResourceManager manager = new PersistenceResourceManager(
                        persistence,<search builders loaded from configuration:
                        See Search Builder component for configuration details>,
                        <id generators loaded from configuration: See ID
                        Generator component for configuration details>);
                } catch(Exception e) {
                        throw new CreateException();
                }
        }

        // set the session context; called by EJB container
        setSessionContext(SessionContext ctx) {
                context = ctx;
        }

        // business methods
        updateResource(Resource resource, String operator)
                throws IllegalArgumentException, ResourcePersistenceException,{
                if(resource == null || operator == null ||
                        operator.trim().equals(""))
                        throw new IllegalArgumentException();
                try{
                        manager.updateResources(resource, operator);
                } catch(ResourcePersistenceException e) {
                        // if the operation fails, rollback the transaction
                        context.setRollbackOnly();
                        throw e;
                }
        }

        //… other methods in the ResourceManager interface will be implemented
        // along a similar pattern to updateResource.

        // an empty constructor
        ResourceBean() {}

        // some empty methods to fill the session bean contract

        public void ejbRemove() {}

        public void ejbActivate() {}

        public void ejbPassivate() {}
}
```

In order to use the session bean implemented above, a client application will need an instance of its local interface, which can be accessed through the local home interface - the concrete implementations of these classes are generated by the EJB container.

ResourceLocalHome.java:

```
package com.acme.resource;
imports javax.ejb.*;
imports com.topcoder.management.resource.*;

public interface ResourceLocalHome extends EJBLocalHome {

        ResourceLocal create(String namespace) throws CreateException;
}
```

ResourceLocal.java:

```
package com.acme.resource;
imports javax.ejb.*;
imports com.topcoder.management.resource.*;

public interface ResourceLocal extends EJBLocal, ResourceManager {}
```

The deployment descriptor informs the EJB container of what enterprise beans are in use and how to manage the transactions.
Since the transaction attribute "required" is used for all business methods of ResourceBean, all method calls will occur within the scope of a transaction. If the client already has an open transaction, that transaction will be used; otherwise, a new transaction will be started.
Deployment descriptor:

```
<ejb-jar>
	<enterprise-beans>
		<session>
			<ejb-name>Resource</ejb-name>
			<home>com.acme.resource.ResourceLocalHome</home>
			<local>com.acme.resource.ResourceLocal</local>
			<ejb-class>com.acme.resource.ResourceBean</ejb-class>
			<session-type>Stateless</session-type>
			<transaction-type>Container</transaction-type>
		</session>
	</enterprise-beans>
	<assembly-descriptor>
		<container-transaction>
			<method>
				<ejb-name>Resource</ejb-name>
				<method-name>*</method-name>
			</method>
			<trans-attribute>Required</trans-attribute>
		</container-transaction>
	</assembly-descriptor>
</ejb-jar>
```

By using container managed transactions, the behavior of the transaction management can easily be tweaked by editing the deployment descriptor, without altering any code. Enabling this pattern of usage is the only new functionality in version 1.1. Since the actual operations performed are identical to those shown in the existing demo, that part of the demo won't be repeated here.

5. Future Enhancements

At the current time, no future enhancements are expected for this component.