Resource Management 1.3 Component Specification

Aspects of the design that are changed in version 1.3 are blue. Additions in version 1.3 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.

Version 1.2 makes provision for the DAO implementation to be able to accommodate the changes coming from the Object Model for Resource Management 1.1 component, which has changed the association of Resource entity to Submission entity from 1 to 1 to 1 to N. The changes revolve mainly around the SQL needed to deal with multiple submission associations.

Note also that the UML has been thoroughly cleaned up and many classes have been removed because they are actually documented in the *Resource Management 1.1* component on which this design depends. Please note that this clean up has been confirmed by the PM as an enhancement.

Version 1.3 :
A resource can have statistics associated with its role in a project. A statistic is a general notion of an object that contains information about the performance of an user having a role in a project. For the upgrade reviewer statistics are the most important ones. However, the update is designed in way that statistics can be extended to other roles as well.

For the purpose of this design and to maintain the generic meaning of the notion, statistics are denoted by a marker interface. An implementation for review statistics is also provided.

Review statistics are of two types:

- History statistics, these store individual statistics for each project that the reviewers were rated for.
- Average statistics. These statistics are computed and associated with a review for easy retrieval from the last configurable number of history statistics. For example this number can be 15.

The computed statistics are also stored in persistence and overwritten on each upgrade because the statistics will be displayed on the frontend and it will be very expensive to have them computed on each request by each viewer.

Average and history statistics are stored in separated tables (review_average_statistics and history_statistics) in persistence.

1.1 Design Patterns

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. AbstractInformixReviewerStatisticsPersistence also uses the TemplateMethodPattern for the same reason.

Different implementation can be plugged into ReviewerStatisticsManagerImpl which uses ***Strategy Pattern***.

1.2 Industry Standards

SQL, JDBC

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 are any submission entries 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 any rows (i.e. there are previous submission entries), and whether the Resource passed to the method has currently any submission entries (its getSubmissions() method returns a non-empty array value).

1. No previous submission(s) and no current submission(s):
    - Do nothing
2. No previous submission(s) and has current submission(s):
    - For each submission associated with Resource:
        - Add a row to resource_submission. Use the second SQL statement in the Add Resource section)
3. Has previous submission(s) and no current submission(s):
    - For each existing row in resource_submission:
        - Remove a row from resource_submission. Use the second SQL statement in the Delete Resource section
4. Has previous submission(s) and has current submission(s):
   Here we will have to do a bit of everything from steps 1 to 3 above. We can break down the submissions overlap as follows: All the current submissions that need to be associated with this Resource will follow step 2 above. All the previous submissions that do NOT overall any current submissions will need to be removed will follow step 3 above. Finally all the previous submissions that overall current submissions will follow this step:
    - For each overlapped entry (i.e. current == previous):
        - Update a row in resource_submission table using the SQL statement:
          ```
          UPDATE resource_submission
          SET modify_user = ?, modify_date = ?
          WHERE resource_id = ? AND submission_id = ?
          ```

[11]
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:

1. 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.
2. 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 =
?
```

3. 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 all associated entries 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, project_phase_id,
 create_user, create_date, modify_user, modify_date)
 VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

If submissions array is not empty, then insert each 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 = ?
```

The above doesn't need to be broken up into two separate queries for version 1.2 since the number of submissions associated is expected to be reasonable and thus the number of rows coming back for this join is expected to be within a reasonable range.

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>);
```

The above doesn't need to be broken up into two separate queries for version 1.2 since the number of submissions associated is expected to be reasonable and thus the number of rows coming back for this join is expected to be within a reasonable range.

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.

1.3.1.17 Create ReviewerStatistics

If reviewerStatistics.statisticsType==AVERAGE, then execute:

```
insert into average_review_statistics (id, accuracy, coverage,
timeline_reliability, total_evaluation_coefficient, reviewer_id,
eligibility_points, project_id, competition_type_id ,
create_user, create_date, modify_user, modify_date) values
(?,?,?,?,?,?,?,?,?,?,?,?,?)
```

Use reviewerStatisticsIDGenerator to generate new ID.

Remove entry in cache with key=AVERAGE_PREFIX+reviewerId

Else, execute

```
insert into history_statistics (accuracy, coverage,
timeline_reliability, total_evaluation_coefficient, reviewer_id,
eligibility_points, project_id, competition_type_id ,
create_user, create_date, modify_user, modify_date) values
(?,?,?,?,?,?,?,?,?,?,?,?,?)
```

Remove entry in cache with key=HISTORY_PREFIX+reviewerId

End If

Finally, Set reviewerStatistics with generated ID, and put it into cache with key=SINGLE_PREFIX+id;

Return reviewerStatistics entity.

1.3.1.18 Update ReviewerStatistics

If reviewerStatistics.statisticsType==AVERAGE, then execute:

```
update average_review_statistics set accuracy=?,coverage=?,
timeline_reliability=?,total_evaluation_coefficient=?,
reviewer_id=?,eligibility_points=?,project_id=?,
competition_type_id=?,create_user=?,create_dat=?,modify_user=?,
modify_date=? where id=?
```

Remove entry in cache with key=AVERAGE_PREFIX+reviewerId

Else execute:

```
update history_statistics set accuracy=?,coverage=?,
timeline_reliability=?,total_evaluation_coefficient=?,
reviewer_id=?,eligibility_points=?,project_id=?,
competition_type_id=?,create_user=?,create_dat=?,modify_user=?,
modify_date=? where id=?
```

Remove entry in cache with key=HISTORY_PREFIX+reviewerId

End If

Finally, put reviewerStatistics into cache with key=SINGLE_PREFIX+id;

Return reviewerStatistics entity.

1.3.1.19 Retrieve ReviewerStatistics

Check if an object exists in cache with key=SINGLE_PREFIX+id, if yes, just return the object.

Find it in average table by executing:

```
select * from average_review_statistics where id=?
```

If result set is NOT empty, then

Populate a ReviewerStatistics entity with data in result set, and put entity into cache with key=SINGLE_PREFIX+id, finally return the entity

End If

Find it in history table by executing:

```
select * from history_statistics where id=?
```

If result set is empty, return null.

End If

Populate a ReviewerStatistics entity with data in result set, and put entity into cache with key=SINGLE_PREFIX+id, finally, return the entity.

1.3.1.20 Delete ReviewerStatistics entity

Get reviewer id by executing:

```
select reviewer_id from average_review_statistics where id=?
```

If result set is not empty:

Remove entry from cache with key=AVERAGE_PREFIX+reviewerId;

Delete entity by executing:
`delete from average_review_statistics where id=?`
Return true;

Get reviewer id by executing:
`select reviewer_id from history_statistics where id=?`
If result set is not empty:
Remove entry from cache with key=HISTORY_PREFIX+reviewerId
Delete entity by executing:
`delete from history_statistics where id=?`
Return true;

Return false;

1.3.1.21 Get history reviewer statistics by reviewer ID

Check if there is an object in cache with key=HISTORY_PREFIX+reviewerId, if yes, just return that object.

Find history statistics by executing:
`select * from history_statistics where reviewer_id=?`
Populate an array of ReviewerStatistics entities with data in result set.

Put array into cache with key=HISTORY_PREFIX+reviewerId.

Return array.

1.3.1.22 Get average reviewer statistics by competition type

Find statistics by executing:
`select * from average_review_statistics where reviewer_id=? and competition_type_id=?`
If result set is empty, return null.

Populate a ReviewerStatistics entity with data in result set.
Return the entity.

1.3.1.23 Get side by side statistics

Find statistics by executing:
`select a.*, b.* from history_statistics, history_statistics b where a.competition_type_id=? and a.project_id=b.project_id and a.reviewer_id=? and b.reviewer_id=?`
“a” represents the first reviewer, “b” represents the second reviewer.

Create two lists firstList and secondList. For each row in result set, populate two ReviewerStatistics entities , entity for the first reviewer is added into firstList and entity for the second reviewer is added into secondList. Create an new SideBySideStatistics entity, set firstList and secondList to SideBySideStatistics;

Return SideBySideStatistics entity.

1.3.1.24 Get reviewer average statistics

Check if there is an object in cache with key=AVERAGE_PREFIX+reviewerId, if yes, just return that object.

Find average statistics by executing:
`select * from average_review_statistics where reviewer_id=?`
Populate an array of ReviewerStatistics entities with data in result set.

Put array into cache with key= AVERAGE _PREFIX+reviewerId.

Return array.

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.

Version 1.2 changes do not affect the transactional flow in any manner.

1.4 Component Class Overview

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.

Version 1.2 Changes:

Please note that all the changes in version 1.2 revolve around the changes to the association multiplicity between Resource and Submission, which used to be 1 to 1 and is not 1 to N (i.e. a resource can be associated with multiple submissions)

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.

ResourceStatistics(interface):

This is the interface for statistics that maintains the generic meaning.

ReviewerStatistics:

This is the entity class that represents the reviewer statistics, it implements ResourceStatistics interface and extends AuditableResourceStructure that provides auditing capabilities.

StatisticsType:

This class represents the statistics type enum. It's extends from Enum class in Typesafe Enum component.

ReviewerStatisticsManager(interface):

This interface defines the contract for managing ReviewerStatistics entity. It provides CRUD operations for ReviewerStatistics and useful methods for external web modules to retrieve an array of ReviewStatistics object, to get a specific ReviewStatistics for a reviewer and competition type and to return all statistics for projects in which the two reviewers competed against one another (had secondary reviewer role).

ReviewerStatisticsManagerImpl:

This class is the default implementation of ReviewerStatisticsManager interface. It holds and instance of ReviewerStatisticsPersistence and delegates all job to ReviewerStatisticsPersistence instance. It provides CRUD operations of ReviewerStatistics entity.

ReviewerStatisticsPersistence(interface):

This interface defines the contract for managing ReviewerStatistics entities in persistence. It provides CRUD operations for ReviewerStatistics entity. Implementation class is expected to have a constructor with exactly one parameter (namespace) which represents the configuration namespace for that class.

AbstractInformixReviewerStatisticsPersistence(abstract):

This class is the base class for informix reviewer statistics persistence implementations. It implements CRUD operations on ReviewerStatistics entities. DBConnectionFactory is used to create database connections. Abstract methods openConnection, closeConnection and closeConnectionOnError are provided to allow child class to manage transaction by itself or not.

InformixReviewerStatisticsPersistence:

This is a review application persistence implementation. It manages transactions by internally. 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.

UnmanagedTransactionInformixReviewerStatisticsPersistence

This class performs exactly the same tasks as InformixReviewApplicationPersistence, 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.

SideBySideStatistics:

This class represents the side by side statistics for two reviewers.

1.5 Component Exception Definitions

There are no custom exceptions defined for this component, but we will reuse the following exceptions from the *Resource Management 1.1* component:

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.

ReviewerStatisticsManagerException:

This exception is thrown when any error occurred during managing ReviewerStatistics entities. It's thrown by ReviewerStatisticsManager implementations.

ReviewerStatisticsPersistenceException:

This exception is thrown when any error occurred during persistence operations of ReviewerStatistics entities. It's thrown by ReviewerStatisticsPersistence implementations.

ReviewerStatisticsConfigurationException:

This exception is thrown when any error occurred during configuration. It's thrown by ReviewerStatisticsManagerImpl and AbstractInformixReviewerStatisticsPersistence.

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.

Version 1.2 also has no impact on thread safety as the change only affects the structure of persistence for submissions and not the actual thread-safety logic for such persistence.

Version 1.3 also has no impact on thread safety as it follows the thread safety rule of previous versions.

2. Environment Requirements

2.1 Environment

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

2.2 TopCoder Software Components

- **DB Connection Factory 1.0:** Used in SQL persistence implementation to connect to the database. Also used by the Search Builder component.

- **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.
- **Resource Management 1.1:** This is the component for which the persistence plugin has been created. This component depends on all the contracts from Resource Management 1.1 as well as the object model defined there to function.
- **Simple Cache 2.**0: This is the component providing caching functionalities for this component.
- **Configuration Manager 2.**1.5: This is the component used to get configuration values for this component.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

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

3.2 Configuration Parameters

AbstractInformixReviewerStatisticsPersistence:

Parameter	Description	Value
DBConnFactoryNamespace	Configuration namespace for DBConnectionFactory.	String, Not null or empty
ConnectionName	DB connection name.	String, Not null or empty
ReviewerStatisticsIDSequenceName	Name of IDGenerator for ReviewerStatistics	String, not null or empty
CacheNamespace	Namespace for cache.	String, not null or empty

ReviewerStatisticsManagerImpl:

Parameter	Description	Value
PersistenceClassName	Class name of persistence implementation.	String, not null or empty
PersistenceNamespace	Configuration namespace for persistence implementation.	String, not null or empty

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.

4.3 Demo:

Unfortunately, as this component cannot really be described in a full customer scenario without including all the other components that this component depends on.
This demo will not be so much a customer-oriented demo but a rundown of the requirements in a demo form. Where it makes sense, examples of what a customer might do with the information are shown. Please note that this demo relies on classes and interfaces from the Resource Management 1.1 component. Specifically we will assume that a manager used in this demo is an implementation of the `ResourceManager` interface as defined in that component.

4.3.1 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.
```

4.3.2 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>);
```

4.3.3 *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”);
```

4.3.4 *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”);
```

4.3.5 *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();
```

4.3.6 *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”);
```

4.3.7 *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;
// add a submission.
resource.addSubmission(submissionId);
// set the users that modified the record.
resource.setModificationUser("me");
// set the timestamp of the modification
```

```
resource.setModificationTimestamp(new Date());
// 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, projected
                    , notificationTypeId
                    , "operator");

// Now I want to delete user 1 from the notification list.
persistence.removeNotification(1, projected
                    , 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});
```

4.3.8 *Manage submissions*

```
persistence.addResourceRole(DBTestUtil.createResourceRole(5));
// Here we will show how to associate many submissions with a
// resource. There are basically three possibilities:

// (1) We have a resource with no submissions
// (i.e. 0 submissions)
int resourceId = 1;

// This will create a new resource with no submissions
Resource resource = DBTestUtil.createResource(resourceId, 1,
1);


persistence.addResource(resource);

// (2) we now add a single submission to the resource
// (i.e. 1 submission)

// we associated a single submission
resource.addSubmission(new Long(121));

// now we will have a resource associated with a single
// submission in the store.
persistence.updateResource(resource);
// (3) Now we will add some more submissions
// (i.e. N submissions)
resource.addSubmission(new Long(122));
resource.addSubmission(new Long(123));

// at this point we have total of 3 submissions associated

// We can print those
Long[] submissions = resource.getSubmissions();
for (int i = 0; i < submissions.length; i++) {
    System.out.println("submission: " + submissions[i]);
}
```

```
        // we can also set the submissions in a single bulk operation
        Long[] bulkSubmissions = new Long[] {new Long(1200), new
Long(1201), new Long(1202), new Long(1203)};
        resource.setSubmissions(bulkSubmissions);
        // note that this will replace all previous submissions.

        // we can also add bulk submissions
        Long[] moreBulkSubmissions = new Long[] {new Long(1204), new
Long(1205), new Long(1206)};
        for (int i = 0; i < moreBulkSubmissions.length; i++) {
            resource.addSubmission(moreBulkSubmissions[i]);
        }
        // this will add to the existing submissions so that we will
        // now have 7 submissions.

        // we do an update with the manager
        persistence.updateResource(resource);

        // we can also remove a submission from the model
        resource.removeSubmission(new Long(1200));
        // and we can remove a bulk of submissions
        for (int i = 0; i < moreBulkSubmissions.length; i++) {
            resource.removeSubmission(moreBulkSubmissions[i]);
        }
        // after this we only have association with 1201, 1202, 1203
        // We can print those
        submissions = resource.getSubmissions();
        for (int i = 0; i < submissions.length; i++) {
            System.out.println("submission: " + submissions[i]);
        }
```

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.topcoder.management.resource.persistence.sql;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
```

```java
import javax.ejb.SessionContext;

import com.topcoder.db.connectionfactory.DBConnectionFactory;
import com.topcoder.db.connectionfactory.DBConnectionFactoryImpl;
import com.topcoder.management.resource.Notification;
import com.topcoder.management.resource.NotificationType;
import com.topcoder.management.resource.Resource;
import com.topcoder.management.resource.ResourceRole;
import com.topcoder.management.resource.persistence.ResourcePersistence;
import com.topcoder.management.resource.persistence.ResourcePersistenceException;

/**
 * <p>
 * Use of <code>{@link UnmanagedTransactionResourcePersistence}</code> in an EJB.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 1.1
 *
 */
public class ResourceBean implements SessionBean, ResourcePersistence {
    /**
     * <p>
     * This is a session context that is used to obtain the run time session context.
     * </p>
     */
    private SessionContext sessionContext;

    /**
     * <p>
     * Represents the unmanaged persistence for demo.
     * </p>
     */
    private transient AbstractResourcePersistence persistence;

    /**
     * Does nothing.
     */
    public void ejbActivate() {
    }

    /**
     * Does nothing.
     */
    public void ejbPassivate() {
    }

    /**
     * Does nothing.
     */
    public void ejbRemove() {
    }

    /**
     * Sets the sessionContext.
     *
     * @param context
     *            the session context
     */
    public void setSessionContext(SessionContext context) {
        this.sessionContext = context;
    }

    /**
     *
     * @throws CreateException
     *             if there was an issue with creation (we rethrow)
     */
    public void ejbCreate() throws CreateException {
        try {
            DBConnectionFactory connectionFactory = new DBConnectionFactoryImpl();
```

```
            persistence = new UnmanagedTransactionResourcePersistence(connectionFactory);
        } catch (Exception e) {
            throw new CreateException("Failed to create UnManagedTransactionPersistence -
        " + e.getMessage());
        }
    }

    /**
     * @see com.topcoder.management.resource.ResourceManager#addNotifications(long[],
        long, long,
     *      java.lang.String)
     */
    public void addNotification(long users, long project, long notificationType, String
        operator)
        throws ResourcePersistenceException {
        try {
            persistence.addNotification(users, project, notificationType, operator);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#addNotificationTy
        pe
     * (com.topcoder.management.resource.NotificationType)
     */
    public void addNotificationType(NotificationType notificationType) throws
        ResourcePersistenceException {
        try {
            persistence.addNotificationType(notificationType);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see com.topcoder.management.resource.persistence.ResourcePersistence#addResource
     * (com.topcoder.management.resource.Resource)
     */
    public void addResource(Resource resource) throws ResourcePersistenceException {
        try {
            persistence.addResource(resource);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#addResourceRole
     * (com.topcoder.management.resource.ResourceRole)
     */
    public void addResourceRole(ResourceRole resourceRole) throws
        ResourcePersistenceException {
        try {
            persistence.addResourceRole(resourceRole);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#deleteNotificatio
        nType
```

```
     * (com.topcoder.management.resource.NotificationType)
     */
    public void deleteNotificationType(NotificationType notificationType) throws
        ResourcePersistenceException {
        try {
            persistence.deleteNotificationType(notificationType);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#deleteResource
     * (com.topcoder.management.resource.Resource)
     */
    public void deleteResource(Resource resource) throws ResourcePersistenceException {
        try {
            persistence.deleteResource(resource);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#deleteResourceRol
        e
     * (com.topcoder.management.resource.ResourceRole)
     */
    public void deleteResourceRole(ResourceRole resourceRole) throws
        ResourcePersistenceException {
        try {
            persistence.deleteResourceRole(resourceRole);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#loadNotification(
        long, long,
     *      long)
     */
    public Notification loadNotification(long user, long project, long notificationType)
        throws ResourcePersistenceException {
        try {
            return persistence.loadNotification(user, project, notificationType);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#loadNotificationT
        ype(long)
     */
    public NotificationType loadNotificationType(long notificationTypeId) throws
        ResourcePersistenceException {
        try {
            return persistence.loadNotificationType(notificationTypeId);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
```

```
}

/**
 * @see
    com.topcoder.management.resource.persistence.ResourcePersistence#loadNotificationT
    ypes(long[])
 */
public NotificationType[] loadNotificationTypes(long[] notificationTypeIds)
    throws ResourcePersistenceException {
    try {
        return persistence.loadNotificationTypes(notificationTypeIds);
    } catch (ResourcePersistenceException e) {
        sessionContext.setRollbackOnly();
        throw e;
    }
}

/**
 * @see
    com.topcoder.management.resource.persistence.ResourcePersistence#loadNotifications
    (long[], long[],
 *      long[])
 */
public Notification[] loadNotifications(long[] userIds, long[] projectId, long[]
    notificationTypes)
    throws ResourcePersistenceException {
    try {
        return persistence.loadNotifications(userIds, projectId, notificationTypes);
    } catch (ResourcePersistenceException e) {
        sessionContext.setRollbackOnly();
        throw e;
    }
}

/**
 * @see
    com.topcoder.management.resource.persistence.ResourcePersistence#loadResource(long
    )
 */
public Resource loadResource(long resourceId) throws ResourcePersistenceException {
    try {
        return persistence.loadResource(resourceId);
    } catch (ResourcePersistenceException e) {
        sessionContext.setRollbackOnly();
        throw e;
    }
}

/**
 * @see
    com.topcoder.management.resource.persistence.ResourcePersistence#loadResourceRole(
    long)
 */
public ResourceRole loadResourceRole(long resourceId) throws
    ResourcePersistenceException {
    try {
        return persistence.loadResourceRole(resourceId);
    } catch (ResourcePersistenceException e) {
        sessionContext.setRollbackOnly();
        throw e;
    }
}

/**
 * @see
    com.topcoder.management.resource.persistence.ResourcePersistence#loadResourceRoles
    (long[])
 */
public ResourceRole[] loadResourceRoles(long[] resourceRoleIds) throws
    ResourcePersistenceException {
    try {
```

```
            return persistence.loadResourceRoles(resourceRoleIds);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#loadResources(lon
        g[])
     */
    public Resource[] loadResources(long[] resourceIds) throws
        ResourcePersistenceException {
        try {
            return persistence.loadResources(resourceIds);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#removeNotificatio
        n(long, long,
     *      long, java.lang.String)
     */
    public void removeNotification(long user, long project, long notificationType, String
        operator)
        throws ResourcePersistenceException {
        try {
            persistence.removeNotification(user, project, notificationType, operator);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#updateNotificatio
        nType
     * (com.topcoder.management.resource.NotificationType)
     */
    public void updateNotificationType(NotificationType notificationType) throws
        ResourcePersistenceException {
        try {
            persistence.updateNotificationType(notificationType);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
     * @see
        com.topcoder.management.resource.persistence.ResourcePersistence#updateResource
     * (com.topcoder.management.resource.Resource)
     */
    public void updateResource(Resource resource) throws ResourcePersistenceException {
        try {
            persistence.updateResource(resource);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }

    /**
```

```
     * @see
       com.topcoder.management.resource.persistence.ResourcePersistence#updateResourceRol
       e
     * (com.topcoder.management.resource.ResourceRole)
     */
    public void updateResourceRole(ResourceRole resourceRole) throws
        ResourcePersistenceException {
        try {
            persistence.updateResourceRole(resourceRole);
        } catch (ResourcePersistenceException e) {
            sessionContext.setRollbackOnly();
            throw e;
        }
    }
}
```

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.topcoder.management.resource.persistence.sql;

import javax.ejb.EJBLocalHome;

/**
 * <p>
 * An EJBLocalHome interface as part of the UnmanagedResourcePersistence demo.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 1.1
 */
public interface ResourceLocalHome extends EJBLocalHome {
    // empty
}
```

ResourceLocal.java:

```
package com.topcoder.management.resource.persistence.sql;

import javax.ejb.EJBHome;

import com.topcoder.management.resource.ResourceManager;

/**
 * <p>
 * EJBHome interface as part of the UnmanagedResourcePersistence demo.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 1.1
 */
public interface ResourceLocal extends EJBHome, 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.

4.2.9 Manage ReviewerStatistics entities

```
// create an instance of ReviewerStatisticsManagerImpl
ReviewerStatisticsManagerImpl manager = new
ReviewerStatisticsManagerImpl(ReviewerStatisticsManagerImpl.class
.getName());

ReviewerStatistics rs = new ReviewerStatistics();
// set properties of ReviewerStatistics
rs.setReviewerId(1);
rs.setProjectId(1);
rs.setCompetitionTypeId(1);
rs.setAccuracy(0.65);
rs.setModificationUser("topcoder");
rs.setModificationTimestamp(new Date(0));
rs.setCreationUser("topcoder");
rs.setCreationTimestamp(new Date(0));
// create an instance of ReviewerStatistics
rs = manager.create(rs);

// create another instance
ReviewerStatistics anotherRS = new ReviewerStatistics();
anotherRS.setReviewerId(2);
anotherRS.setProjectId(1);
anotherRS.setCompetitionTypeId(1);
anotherRS.setAccuracy(0.8);
anotherRS.setModificationUser("topcoder");
anotherRS.setModificationTimestamp(new Date(0));
anotherRS.setCreationUser("topcoder");
anotherRS.setCreationTimestamp(new Date(0));
anotherRS = manager.create(anotherRS);
```

```
// update ReviewerStatistics
rs.setAccuracy(0.85);
rs = manager.update(rs);
// accuracy is set to 0.85

// retrieve ReviewerStatistics
ReviewerStatistics rs2 = manager.retrieve(rs.getId());
// rs2 has the same content as rs
assertEquals(rs.getId(), rs2.getId());

// delete rs
manager.delete(rs.getId());
// rs should be deleted

// create an instance of ReviewerStatistics
rs = manager.create(rs);

// get reviewer statistics by competition type
ReviewerStatistics rs3 =
manager.getReviewerStatisticsByCompetitionType(1, 1);
// rs3 should be the same as rs, its reviewer id is 1 and
// competition type id is 1.

// get Side By Side Statistics
SideBySideStatistics rs4 = manager.getSideBySideStatistics(1, 2,
1);
// an array with 2 elements is returned, one is for reviewer with
// id=1, the other is for reviewer with id=2

// get Reviewer Average Statistics
ReviewerStatistics[] rs5 =
manager.getReviewerAverageStatistics(2);
// an array with 1 element is returned, average statistics for
// reviewer 142 with competition type=1

// get Reviewer history Statistics
ReviewerStatistics[] rs6 = manager.getReviewerStatistics(2);
// an array with 10 elements is returned, all elements have
// reviewer id=2 and competition type id=2;
```

5. Future Enhancements

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