Time Tracker Rates 3.2 Component Specification

1. Design

Within the Time Tracker application, “Rates” are stored to assist in the calculation of values used by the application. As the data for multiple companies is stored by the application, each “Rate” is split into two conceptual halves: Firstly, a common rate, which is simply a description and an ID - e.g. “Overtime per hour”. Then, each rate is combined with a company, storing also the value which the rate takes - e.g. “Company XYZ Overtime per hour = $50”. Different companies then can all share the same rate description, but have different values to each other.

This component manages the “Rate” objects described above - that is, rate descriptions always coupled with company identification. These Rate objects can be added, updated and removed from a Rate Manager, as well as retrieved by Company Id as well as other possible criteria.

The Rate Manager itself is backed by a pluggable persistence layer, to make sure the Rate information is stored between application runs - the type of persistence is set at configuration, and a default Informix version is provided with this component.

Finally, whenever insert/update/delete operations are performed (either on single or multiple rates), the ability is provided to also persist audit records detailing each change to the database.

Please note that changes to version 3.1 will be shown in **red** (new entries) and **blue** (modification of existing entries).

In version 3.2 we will add number of enhancements/clarification to the 3.1 version

1. We will create a mid-tier layer for transactional control, which will use the stateless Session Bean technology from J2EE. The basic premise is that the RateManager will now act as a delegate, which will talk to the EJB (which is controlled/observed by the J2EE container), which will in turn talk to the DAO**. The important thing to remember here is that the EJB will participate in transactions that are delineated and controlled by the container**.
2. It was also decided that usage of Configuration Manager is safe if properly initialized on the server (since it is a singleton) since at that time (when the EJB is created and used in the container) all the namespace data would have been loaded into memory.

1.1 Design Patterns

Delegate + Strategy: The Audit Manager class loads a pluggable persistence layer, then delegates all of the methods to this persistence strategy.
The **Business Delegate** pattern (J2EE pattern) is used by the RateManager to delegate all actions to the EJB (which then ferries that to the proper DAO)
Dao pattern is used to separate persistence implementation/logic and the controller (manager/ejb)

1.2 Industry Standards

SQL is used by the provided Informix persistence layer to persist information to the database connected.

We also use JDBC, J2EE, and EJB

1.3 Required Algorithms

Most complex algorithms within the component are displayed on sequence diagrams - the most involved - add Rates (update and delete are similar) - is also shown below:

```
addRates(rates):
    Obtain a connection to the database.
    Prepare an SQL_INSERT_RATE statement
    For each rate in rates:
        Insert the rate information parameters
        statement.addBatch();
    statement.executeBatch();
    For each result:
        If the update failed: Log at ERROR level
        If the update succeeded:
            auditAction(null*, rate, "INSERT"):
                create Audit Header
                insert header information (see appendix)
                for each changed rate detail, create Audit Detail
                insert audit into the action
```

**If this is an update or delete statement, then retrieveRate(rateid, companyid, false) should first be called and the result passed here as the old rate information. See appendix*

1.3.1 Transactional Considerations

All transactional control is now in the J2EE container, which means that we do not control the transaction explicitly through the connection (we cannot actually as this is forbidden) What we do is we rollback transactions when there is an issue through the `setRollbackOnly()` method is `SesssionContext` which then signals to the container to control the transaction properly.
There is an additional aspect, which needs to be considered: When we are dealing with AuditManager we only use the manager to create audits but we do not use it to delete audits. Again here we let the container UNDO any audit issues.

1.4 Component Class Overview:

RateManager:
A RateManager is the entry point into the storage and retrieval of rates within the system - it provides the basic CRUD methods, for both single and multiple Rates. Each method within is delegated to a pluggable persistence layer, the type of which is determined via configuration.

Rate:
Bean class that stores information about a Rate within a Time Tracker application. This extends the TimeTrackerBean base, so contains the common ID as well as creation and modification information. In addition to this, information is stored containing the description of the rate, the value the rate takes, as well as a company the rate is for. As it is a Bean class, it has a no-argument constructor and all members have appropriately named get/set methods. In addition, no parameter checking is performed; the validation of parameters is left to the web application.

RatePersistence << interface >>:
Interface defining the necessary methods for any pluggable persistence layer for use with the Rates component. Provided are four basic CRUD operations for batches of rates

(add/update/delete/retrieve Rates) as well as two additional methods that allow single rates to be obtained, identified by company and ID or description. It is up to each implementation to define how much information it loads as well as when it throws exceptions, please read any implementing class's documentation for more details. Each implementing class should be able to be constructed by using the Object Factory component. In addition, it is assumed that each implementing class will handle its own thread safety.

InformixRatePersistence:
The InformixRatePersistence class is the default persistence plug-in for this component - it stores the rate information inside Informix tables (the ERD has been provided). Implementing the RatePersistence interface, it provides the basic CRUD operations. In addition, two private utility methods have been provided, to help load rates from a database row, as well as to auditing any insert/delete/update action.

LocalRate
This is a simple local home interface used to create a local bean for the rate persistence delegation actions.

LocalHomeRate
This is a local interface for the contract for CRUD operations on rate data. This interface will be used in implementing a stateless session ejb with container manager transactions.

RateEjb
This is an implementation EJB of the LocalRate interface. This is nothing more than a delegating wrapper around the RatePersistence instance but with an interception of exceptions from the dao and with subsequent transaction control (rollback only) when an issue has been detected/signaled.
In other words this acts like a container based transactional control plugin. This is a stateless session bean with a container manager transaction control. This ejb will join any started transactions.

1.5 Component Exception Definitions

RatesManagerException:
Exception used by the Rates component whenever there is a problem within a RateManager instance. For example, if its pluggable persistence encounters an error, that will be wrapped and thrown as a RateManager exception. Two constructors are provided that both take a reason why the error occurs, and one taking the error which caused this to be thrown.

RatesPersistenceException:
Exception used by the Rates component whenever there is a problem within the pluggable persistence layer. Any class implementing the RatePersistence interface may throw this if errors are encountered while persisting information, for example caused by SQL or IO exceptions.

RatesConfigurationException:
Exception used by the Rates component whenever there is a problem while initializing all the objects based on configuration parameters. For example, if required parameters are missing or invalid, then this exception is thrown. It is also recommended any implementation of RatePersistence throws this exception.

javax.ejb.CreateException: if there was an issue with creation of an ejb.

1.6 Thread Safety

The Java bean within the component - Rate - is not threadsafe - there is no protection against concurrent read/writes of the members. Due to its intended usage, it is assumed that it will be used in a thread-safe manner; synchronization is not performed in the interest of speed and simplicity.
All implementations of the persistence interface should handle their thread safety, and the provided Informix version does this by having immutable members (log and connection name) read on construction, and not modifying the remaining members - the interactions performed on the database utilize a PreparedStatement's batch transactions, any failures are logged.
The remaining class - RateManager - is thread safe as it delegates all method calls to its persistence plug-in, which is assumed to be thread safe.

The newly introduced transaction control (version 3.2) via EJB Stateless Session Bean (container managed) makes this a more stable implementation since now we do not have to worry about what other managers (who might be working on same data) are doing. The actual control will be done based on transaction participation, which will be controlled by the container. Thus we do not have to worry about this aspect any more.

2. Environment Requirements

2.1 Environment

- Java 1.4 for development
- Java 1.4 and 1.5 for compilation

2.2 TopCoder Software Components

- Base Exception 1.0 for the Base Exception superclass of custom exceptions.
- Configuration Manager 2.1.5 for reading parameters from configuration.
- DB Connection Factory 1.0 for configurable Connection loading.
- Logging Wrapper 1.2 for access to configurable Log instances.
- Object Factory 2.0.1 for loading of the pluggable persistence layer.
- Time Tracker Audit 3.2 for access to the audit manager – Here we assume that this version will be available for this component since the transaction control must be done for all managers in the same way.
- Time Tracker Company 3.2 for access to company instances. Here we assume that this version will be available for this component since the transaction control must be done for all managers in the same way.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.rates
com.topcoder.timetracker.rates.persistence
com.topcoder.timetracker.rates.ejb

3.2 Configuration Parameters

3.2.1 Rate Manager configuration properties

Parameter	Description	Values
objectFactoryNamespace (required)	Namespace where the object factory's configuration is stored.	Valid Object Factory namespace.
persistenceObject (required)	Object factory key used to load the AuditPersistence implementation.	Valid Object Factory persistence key.
persistenceNamespace/ *connectionName* (optional)	Name of connection to acquire from DB Connection Factory - if not provided, a default connection is used.	Valid Connection Factory connection name.
persistenceNamespace/ *logName* (optional)	Name of log to acquire from the Log Factory - if not provided, a default log is used.	Valid Log Factory log name.
objectFactoryNamespace (required)	Namespace where the object factory's configuration is stored.	Valid Object Factory namespace.
persistenceNamespace/ *connectionFactoryKey* (required)	Object factory key used to load the Connection factory implementation within the persistence layer.	Valid Object Factory persistence key.
persistenceNamespace/ *auditManagerKey* (required)	Object factory key used to load the Audit manager implementation within the persistence layer.	Valid Object Factory persistence key.
rate_local_ejb_reference_name	This is an actual JNDI property name as used in the deployment descriptor for the EJB so that we can obtain a JNDI initial context. This is basically the value that we would see in the deployment descriptor by which the bean will be bound to JNDI name. ***Required.***	"rate.rateLocalHome"

A sampleConfig.xml file has been provided with demonstrations of each parameter, located at the RateManager's default namespace.

3.2.2 EJB Configuration Entries (via deployment descriptor)

Parameter	Description	Values
of_namespace	An optional Object Factory namespace used to specify the namespace used within Object Factory configuration. Default namespace is assumed if not provided. ***Optional.***	of_namespace
of_rate_persistence_key	This is an optional (by default we assume that the InformixRatePersistence will be used if nothing is specified) Object Factory key, which would be used to obtain an instance of the RatePersistence. ***Optional.***	of_rate_persistence_ key

3.3 Dependencies Configuration

Many components listed in 2.2 will require some form of configuration to allow the configuration of this component to be used - e.g. connections and logs set up. Please see their component specification for more details.

We also assume that the EJB has been properly configured which would include:

- Proper EJB deployment descriptor creation
  - We need to ensure that the proper configuration name for DAO has been included so that it can be fetched through the InitialContext in the EJB

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

See below for example usage:

4.3 Demo

Note that for version 3.2 there is no change to the demo since the user is not aware of the changes.

```
// Create a new manager, using default namespace
RateManager rateman = null;
try{ rateman = new RateManager(); }
catch(RateConfigurationException rce){
   return 0; // hopefully shouldn't occur
}

// Create a new rate for company #3, type #21
Company myCompany = ...; // retrieve from application
myCompany.setId(3);

// Note - no description set, null is default.
Rate myRate = new Rate();
myRate.setId(21);
myRate.setCompany(myCompany);
myRate.setRate(50.0);

// Persist:
try{ rateman.addRate(myRate, true); } // fails, so no audit
catch(RateManagerException rme){
   // this should be thrown, as null descriptions
   // wont be allowed using the default Informix persistence
   System.out.println("Must have non-null description!");
}

// Add description and persist within batch
myRate.setDescription("Overtime per hour");
Rate rates[] = new Rate[1]; rates[0] = myRate;
rateman.addRates(rates, true); // 1 Audit header added

// Double the rates to please the staff, persist the change
myRate.setRate( myRate.getRate() * 2.0 );
rateman.updateRate(myRate, false); // Don't audit

// retrieve the rate:
Rate[] companyRates = rateman.retrieveRates(myCompany.getId());
assert(companyRates.length == 1 && companyRates[0].getId() == 21);

// remove the rate (in batch) then try to retrieve it:
```

```
rateman.removeRates(rates, true); // 1 Audit header added
assert(rateman.retrieveRate(21, 3) == null); // no match, null returned.
assert(rateman.retrieveRate(“Overtime per hour”, 3) == null); //  ‘’

// Note - in total, 2 new Audit header records should be added
```

5. Future Enhancements

One possible enhancement would be to provide a better range of search options - for example, taking a generic search filter and each matching rate is returned.
Another addition might be to allow rates to have company instances loaded, rather than null being used.
Finally, an enhancement that may be useful later is a different persistence plug-in, for example using XML files or a different database provider.

6.

Appendix:

The Entity Relationship Diagram (ERD) for the Rates tables (rate and comp_rate) has been provided with the submission. Each Rate member should map simply to a column within the database.

For the Audit Headers, the following information is used *(constants in italics)*

Member	Column
id	*Not used - audit manager provides this*
entityId	Rate.getId()
creationDate	Rate.getModificationDate()
tableName	*"comp_rate"*
companyId	Rate.getCompany().getId()
creationUser	Rate.getModificationUser()
actionType	*"INSERT"/"UPDATE"/"DELETE"*
clientId	*0*
projectId	*0*
resourceId	*0*
applicationArea	*ApplicationArea.TT_CONFIGURATION*

And for Audit Details, the relevant old/new values are obtained from the previous/new Rate values, with the column name taken from the appropriate ERD column.

The audit itself is added with code similar to:

```
void auditAction(prev, curr, action){
   AuditHeader header = new AuditHeader();
   header.setEntityId(curr.getId());
   header.setCreationDate(curr.getModificationDate());
   // ... etc, see above for uses

   // for each column that has changed:
   ArrayList details = new ArrayList();

   if(prev == null || curr == null || prev.getRate() != curr.getRate()){
      AuditDetail detail = new AuditDetail();
      detail.setColumnName("rate");
      detail.setOldValue(prev == null ? null : "" + prev.getRate());
      detail.setNewValue(curr == null ? null : "" + curr.getRate());
      details.add(detail);
   }

   if(prev == null || curr == null || prev.getRate() != curr.getRate()){
      AuditDetail detail = new AuditDetail();
      detail.setColumnName("description");
      detail.setOldValue(prev == null ? null : prev.getDescription());
      detail.setNewValue(curr == null ? null : curr.getDescription());
      details.add(detail);
   }
   // etc. for each column in the comp_rate table.

   header.setDetails( details as an AuditDetail[] );
   auditManager.createAuditRecord(header); // send record for auditing.
```