Time Tracker Notification Component Specification

1. Design

The Notification custom component is part of the Time Tracker application. It provides an abstraction of the notification features used to communication via email to users of the system. This component integrates the scheduling, sending and manages the persistence and other business logic required by the application. NotificationManager is the main class of this component. It groups all the methods of NotificationPersistence and NotificationSender interfaces for easy access. Application users can get, create, update, delete and search the notification in the persistence, as well as send the notification manually.

Version 3.2 Changes

Version 3.2 addresses the Transaction Management strategy of the Time Tracker application. This is done by utilizing an EJB container to manage the operations being performed by the different Time Tracker components.

The design takes the approach of building an EJB layer on top of the existing classes, rather than modifying the old design. Building the layer has the advantage of allowing the user the easy option of switching out of EJB, and moving to a different Transaction Management strategy should the need arise. The EJB layer is composed of a Business Delegate, the Local and LocalHome interfaces, and a Stateless SessionBean. The Business Delegate is responsible for looking up the LocalHome interface and obtaining an instance of the Local interface. It will then use the Local interface to delegate business calls to the SessionBean. The SessionBean itself has no business logic other than managing the transactions - the actual business logic exists in the old classes of the existing design.

The following changes were done:

- NotificationPersistenceDelegate, NotificationPersistenceLocalHome, NotificationPersistenceLocal and NotificationPersistenceSessionBean classes were all added in a new ejb subpackage. This comprises the EJB layer of the component.
- A new Sequence Diagram was added to depict the functionality within the EJB layer.
- Another Sequence Diagram was added to depict the functioning of the NotificationEvent as requested in the Software Development Forums.
- A note was added to the Use Cases, saying that the application may now run under a J2EE container. Since the requirements for the new changes are not actual business functionality, but rather more on the inner functionality of the component, it did not seem appropriate to introduce a new use case for the new requirements.
- There was no need to refactor the DAOs, since no business logic was found in the DAOs. The JBoss Transaction DataSource may be configured using ConnectionFactory.
- The design attempts to adhere to the J2EE specification of not allowing File access by delegating all file access to an external NotificationPersistenceFactory class. This is a pragmatic approach which has been proven to work in previous designs (see the Orpheus Application components, like Game Persistence, Administration Persistence components). File access does not occur within the

SessionBean itself, but rather within external classes when the application is called. It is also possible for File Access to occur before any SessionBean calls occur, since the ConfigManager may be initialized beforehand.

- Note that the only other alternative to not using ConfigManager and configuration files will be to restrict usage of ANY TC components that utilize configuration files - these include the Connection Factory, Search Builder and ID Generator components, which are a core part of the given component.
- This upgrade also makes a slight modification to the API itself by adding a FilterFactory class to the NotificationPersistence. Such a modification will make it much more convenient for the users to perform their searches.

1.1 Design Patterns

NotificationPersistence and its implementation implement the strategy pattern for NotificationManager. NotificationSender and EmailNotificationSender also implement this pattern.

NotificationPersistence and its implementation implement the DAO pattern

NotificaionManager implements the façade pattern.

NotificationFilterFactory and NotificationPersistenceFactory implement the Factory pattern.

NotificationPersistenceDelegate implements the Business Delegate pattern.

1.2 Industry Standards

JDBC

Informix

EJB

1.3 Required Algorithms

Developers must be familiar with the JDBC and the schema defined in TimeTrackerClient_Notification_ERD.jpg.

1.3.1 Transaction of InformixNotificationPersistence

All the methods of InformixNotificationPersistence that involve in the database operations, should put the database queries in one transaction. Take the updateNotification (notification) method for instance, in order to update notification, this method will also be responsible for updating the associated notification_clients, notification_resources and notification_projects. All these database operations should be put in one transaction. The connection is retrieved from DBConnectionFactory, and it should be closed after operation to release to connection pool.

Note: As of Version 3.2, this portion of the Algorithm Spec does not apply. From the EJB Specification, it is forbidden to use JDBC Transactions within a Container-managed transaction, so the DAO must NOT use transactions.

1.3.2 *Validate Notification in InformixNotificationPersistence*

When creating, updating the notifications in the Informix persistence, the Notification instances should be validated against the database schema (refer to TimeTrackerClient_Notification_ERD.jpg). For instance, in the schema, the notification table must have non-null subject field, then the Notification instance's subject field must be set before inserted into the database. For invalid client instance, throw NotificationPersistenceException.

1.3.3 *Create, update and delete Notification*

For creating operation, the notification id is generated by IDGenerator. The notification_clients, notification_resources and notification_projects should be updated. If the client to create already exists, throw NotificationPersistenceException. If there are duplicate schedule id associated with the new notification id, throw NotificationPersistenceException. If the auditing functionality is enabled, all the created columns in notification table should be audited.

For the update operation, developers can first delete the mapping in notification_clients, notification_resources and notification_projects tables and then insert the new mapping specified in Notification instance. If the auditing functionality is enabled, all the modified columns in notification table should be audited. If there are duplicate schedule ids associated with the notification id, throw NotificationPersistenceException.
If any error occurred, must rollback the auditing entry. Note that rolling back is not explicitly needed in version 3.2; The EJB container will automatically roll back the audit so long as the setRollbackOnly() method is signalled.

For the delete operation, this operation is responsible for delete the notification table as well as the mappings defined in notification_clients, notification_resources and notification_projects tables. If the give notification id does not exist in the notification table, throw NotificationPersistenceException. If the auditing functionality is enabled, all the deleted columns in notification table should be audited.

The three operations will only effect the notification, notification_resources, notification_clients and notification_projects tables.

Set AuditHeader in this way:

The short descriptions of each field in AuditHeader:

entityId: Notification.getId()

tableName: notification

companyId: user.getCompanyId()

actionType: INSERT, DELETE or UPDATE depending of the action type

applicationArea: TT_NOTIFICATION

resourceId: not set

creationUser : username

For the create operation, the old value of the audit detail is null.

For the delete operation, the new value of the audit detail is null.

1.3.4 Search Notifications with filter.

This component will provide search functionalities based on a logical (AND, OR, NOT) combination of search filters. The return will be a collection of Notification objects. If no results are found then the result set will be empty. An exception will be thrown in the event of an error condition being raised. The following is a summary of the required filters:

- Return all entries with a given company Id
- Return all entries with a given project Id
- Return all entries with a given client Id
- Return all entries with a given resource Id
- Return all entries with a given status
- Return all entries with last sent within a given inclusive date range (may be open-ended)
- Return all entries with next send within a given inclusive date range (may be open-ended)
- Return all entries with Subject line containing some text
- Return all entries with Message line containing some text
- Return all entries with from line containing some text
- Return all entries created within a given inclusive date range (may be open-ended)
- Return all entries modified within a given inclusive date range (may be open-ended)
- Return all entries created by a given username
- Return all entries modified by a given username

Please note that the default filter field is the column name. For example, if users try to search the notification whose project id is 1, the filter will be new EqualFilter("project_id" ,new Long(1)); however, final users can change the mapping in the SearchBundle. The context string for SearchBundle is:

select [fields] from notification, notification_resources, notification_clients, notification_projects

where notification.notification_id=notification_resources.notification_id and notification.notification_id=notification_clients.notification_id and notification.notification_id=notification_projects.notification_id and

The [fields] should contain all the necessary attributes for Notification

The developer should note that a NotificationFilterFactory was introduced in version 3.2. The developer should provide filter name mappings accordingly:

- PROJECT_ID_NAME -> notification_projects.project_id
- COMPANY_ID_NAME -> notification_projects.company_id (may not be present in the schema - but the schema should be updated to contain this)
- CLIENT_ID_NAME -> notification_clients.client_id
- RESOURCE_ID_NAME -> notification_resources.notification_id
- ACTIVE_NAME -> notification.status
- LAST_SENT_NAME -> notification.last_time_sent
- NEXT_SEND_NAME -> notification.next_time_send
- FROM_LINE_NAME -> notification.from_line
- MESSAGE_NAME -> notification.message
- SUBJECT_NAME -> notification.subject
- CREATION_USER_NAME -> notification.creation_user*

- MODIFICATION_USER_NAME -> notification.modification_user*
- CREATION_DATE_NAME -> notification.creation_date*
- MODIFICATION_DATE_NAME -> notification.modification_date*

* - Not present in the schema diagram, but this is assumed, since it is that way with all the other tables.

1.3.5 Send Notification

This algorithm is used by NotificationSender. In order to populate the email message, all the columns in notification table should be selected. To get the contact name and email addresses for user, project and client, the algorithm should first select user account ids, client ids and project ids from notification_resources and notification_projects tables. And get the contacts from ContactManager by these ids. Finally, get the contact name and email address from each contact.

Please note that the TCSEmailMessage's email body should be generated by MessageBodyGenerator with contact name and the message retrieved from notification table.

Finally, use EmailEngine to send the tcs message.

1.4 Component Class Overview

Notification:

The notification bean holds the necessary information to send the email.

This class is mutable and not thread safe.

NotificatonManager:

NotificationManager groups all the methods of NotificationPersistence and NotificationSender interfaces for easy access. Application users can get, create, update, delete and search the notification in the persistence, as well as send the notification manually.

This class is thread safe since it's immutable.

As of Version 3.2, a FilterFactory was added to this class.

NotificationPersistence:

This interface defines the contract of managing the Notification entity in some persistence. It provides the persistence functionalities to create, update and search Notification instances.

The implementation is required to be thread safe.

As of Version 3.2, a FilterFactory was added to this class.

InformixNotificationPersistence:

InformixNotificationPersistence provides the functionality of managing the Notification entity in the Informix database.

All the methods of InformixNotificationPersistence that involve in the database operations, should put the database queries in one transaction. Take the updateNotification (notification) method for instance, in order to update notification, this method will also be

responsible for updating the associated notification_clients, notification_resources and notification_projects. All these database operations should be put in one transaction. The connection is retrieved from DBConnectionFactory, and it should be closed after operation to release to connection pool.

This class is thread safe as long as the inner SearchBundle is not modified externally.

As of Version 3.2, a FilterFactory was added to this class.

NotificationSender:

This interface defines the contract of sending the notification in some way like via email.

The implementation is required to be thread safe.

EmailNotificationSender:

EmailNotificationSender is used to send the notification via email. The notification content is retrieved from database. MessageBodyGenerator is employed to generate the email message body based on the contact name and notification message from database.

This class is thread safe since it's immutable.

MessageBodyGenerator:

This interface defines the contract of generating the email body from the contact name and given message.

The implementation is required to be thread safe.

NotificationEvent:

NotificationEvent will be executed by the Job. It will parse the notification id from the job name and then invoke NotificationManager to send the notification.

This class is not thread safe since its mutable.

NotificationFilterFactory:

A FilterFactory used to create Filters for searching the Notifications.

StringMatchType

An enum used to specify the type of String Matching when building filters.

NotificationPersistenceFactory

A Factory used to create NotificationPersistence instances.

NotificationPersistenceDelegate

This is a business delegate for NotificationPersistence.

NotificationPersistenceSessionBean

This is a SessionBean for NotificationPersistence.

NotificationPersistenceLocalHome

This is the LocalHome interface for NotificationPersistence.

NotificationPersistenceLocal

This is the Local interface for NotificationPersistence.

1.5 Component Exception Definitions

IllegalArgumentException

This exception is thrown in various methods if null object is not allowed, or the given string argument is empty. Refer to the documentation in Poseidon for more details.

NOTE: Empty string means string of zero length or string full of whitespaces.

NotificationPersistenceException

This exception will be thrown by NotificationPersistence and its implementation if any error occurred during persisting the notification entity.

NotificationConfigurationException

This exception will be thrown if failed to create the object from configuration file..

NotificationSendingException

This exception will be thrown by NotificationSender and its implementation if any error occurred during sending the notification.

MessageBodyGeneratorException

This exception will be thrown by MessageBodyGenerator and its implementation if any error occurred during generating the message body.

1.6 Thread Safety

This component is not completely thread safe. The bean class Notification has set and get methods, and are not thread safe. NotificationPersistence, NotificationSender, MessageBodyGenerator and their implementations are thread safe, all of their method can be invoked in a multi-thread environment. NotificationEvent is not thread safe, since it's immutable. To be thread safe, all the methods of NotificationEvent must be properly synchronized externally.

The EJB classes are either stateless or have their state initialized during construction. They are all Thread-safe. The NotificationFilterFactory class is also thread-safe due to being immutable.

The NotificationPersistenceFactory should also perform synchronization when retrieving the NotificationPersistence, to ensure that only a single instance is created and returned.

2. Environment Requirements

2.1 Environment

Java 1.4 or higher.

2.2 TopCoder Software Components

Base Exception 1.0

The custom exception extends BaseException in this component.

Configuration Manager 2.1.5

It is used to load the configuration value

DB Connection Factory 1.0

It is used to get named connection.

Search Builder 1.3.1

It's used to generate the complicated search conditions..

ID Generator 3.0

It is used to get the long id for client entities

Time Tracker Audit 3.2

It is used to audit the notification modification in the database

Time Tracker Common 3.2

Notification extends from TimeTrackerBean from this component.

Time Tracker Contact 3.2

It's used the get the contact instances.

Object Factory 2.0.1

It's used to create the configured object.

Email Engine 2.0

It's used to send email.

Email Address Validator 1.1

Not used, it will be removed in the development phase.

Job Scheduling 3.0

NotificatonEvent implements the ScheduledEnabled from this component.

Job Processor 3.0

Not directly used.

Database Abstraction 1.1

CustomResultSet comes from this component.

Logging Wrapper 1.2

It's used to log the error occurring during sending email notification.

JNDI Context Utility 1.0

Used for looking up and loading the LocalHome interface.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.notification

com.topcoder.timetracker.notification.persistence

com.topcoder.timetracker.notification.ejb

com.topcoder.timetracker.notification.send

3.2 Configuration Parameters

For NotificationManager

Parameter	Description	Values
Of_namespace	The namespace used to create ObjectFactory object. Required.	Must be non-empty string.
persistence_key	The key used to create NotificationPersistence object from ObjectFactory. Required.	Must be non-empty string.
send_notification_key	The key used to create NotificationSender object from ObjectFactory. Required.	Must be non-empty string.

For NotificationPersistenceDelegate

Parameter	Description	Values
context_name	The context name used to retrieve the context from JNDIUtils. If not specified, the default context is used. **Optional.**	Must be non-empty string.
location	The location within the context where the local home should be retrieved. **Required.**	Must be non-empty string.

3.3 Dependencies Configuration

SearchBuilder, DBConnectionFactory and IDGenerator should be properly configured to make this component work.

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

Preload the configuration file into Configuration Manager. Follow demo.

4.3 Demo

See ejb-jar.xml for the deployment descriptor for the EJBs.

4.3.1 Manage Notifications

```
// NotificationManager groups all the methods of NotificationPersistece
// and SendNotification, the demo only show how to use the manager.
// create the manager, assume that namespace, persistence, and
// NotificationSender
// are defined
NotificationManager manager = new NoticiationManager(namespace);
// or create from arguments
manager = new NotificationManager(persistence,sendNotification);

// assume that there is one nitification with id 1 in the database
// get the notification
Notification notification = manager.getNotification(1);

// update the notification
notification.setSubject("hello yyy");
manager.updateNotification(notification);

// create notification
Notification temp = new Notification();
temp.setCompanyId(1);
temp.setSubject("hello xxx");
temp.setFromAddress("xxx@topcoder.com");
temp.setMessage("come here");
temp.setNextTimeToSend(new Date(new Date().getTime() + 1000*1000));
// assume that client 1 exists in the database
temp.setToClients(new long[]{1});
temp.setActive(true);
manager.createNotification(temp);

// search the notification whose client id is 1 or subject is "hello yyy");
NotificationFilterFactory filterFactory =
 manager.getNotificationFilterFactory();


Filter filter1 = filterFactory.createClientIdFilter(1);
Filter filter2 = filterFactory.createSubjectFilter("hello yyy",
  StringMatchType.EQUALS);
Filter orFilter = new OrFilter(filter1, filter2);
// two elements expected
Notification[] ns = manager.searchNotifications(orFilter);

// get all notifications
ns = manager.getAllNotifications();
```

4.3.2 Create a BusinessDelegate to run the SessionBean

```
NotificationPersistence delegate = new
    NotificationPersistenceDelegate("coder.timetracker.notification");

// create a new manager using the delegate. We assume the sender has been
// initialized.
NotificationManager delegatedManager =
    new NotificationManager(delegate, sender)

// Once manager has been initialized, all the functions are similar to section
// 4.3.1 and 4.3.2
```

5. Future Enhancements

None