Time_Tracker_Project 2.0 Component Specification

1. Design

The Time Tracker Project custom component is part of the Time Tracker application. It provides an abstraction of projects and the clients that the projects are assigned to. This component handles the persistence and other business logic required by the application.
The manager classes handle the business logic. They provide a sort of facade for the user to the functionality offered by this component: client management and project management. The following classes implement the model part of this application: Client, ProjectWorker, Project, ProjectManager.
The persistence layer is abstracted using an interface. This allows easy swapping of the persistence storage without changes to the rest of the component. The default implementation uses an Informix database for storage.
An observation: in this release a project can have only one client and only one manager(this was decided on the forum).

The 1.1 version of component provides several enhancements, but maintains the same API.

The 2.0 version of component added new fields in model part, changed all the table names in the database persistence layer, added new constraints in the business logic. The places, where something was changed will be marked by blue color, and the places, where something new was added will be marked with a red color.

Batch Operations
The batch versions of the CRUD (Create/Read/Update/Delete) operations of the persistence layer are provided. This means they accept an array rather than single instances. By caller's choice, the batch operations can be made atomic (all-or-nothing).

Project Search
This functionality adds the ability to search for all projects based on some criteria. The search criteria can be a combination (with use of logical "AND", "OR" and "NOT" operations) of different search filters. The search functionality is very configurable and flexible. The sample configuration file provides an ability of search within the following project parameters:

- Project company Id
- Project name
- Project description
- Project manager
- Project worker
- Project client
- User, who has created the project
- User, who has modified the project
- Project start date
- Project end date
- Project creation date
- Project modification date

Client Search
This functionality adds the ability to search for all clients based on some criteria. The search criteria can be a combination (with use of logical "AND", "OR" and "NOT" operations) of different search filters. The search functionality is very configurable and flexible. The sample configuration file provides an ability of search within the following

client parameters:

- Client company Id
- Client name
- User, who has created the client
- User, who has modified the client
- Client creation date
- Client modification date

Introduced company id and validations regarding to this new attribute.

1.1 Design Patterns

The **strategy** pattern is used for abstracting the persistence implementation in the TimeTrackerProjectPersistence interface.
The **data value object** pattern is used in the Client, Project, ProjectWorker and ProjectManager classes. These classes facilitate the data exchange between the business logic and the persistence layer.
The **composite** pattern is used by the classes representing search filters (particularly by BinaryOperationFilter and NotFilter classes)

1.2 Industry Standards

SQL, JDBC

1.3 Required Algorithm

The SQL statements are very simple select, insert, update and delete queries, so it is not necessary to provide them as algorithms.

The 1.1 version just reuses the SQL statements for CRUD operations from version 1.0, so they are not included. But the SQL statements for search functionality are included in the sample configuration files.

The 2.0 version renamed all the tables and fields from tables as in required specification.

1) An algorithm for the constructor of ProjectPersistenceManager.

```
//read the configuration file
ConfigManager config = ConfigManager.getInstance();
String persistenceClassName = config.getString(namespace, "persistence_class");
String dbNamespace = config.getString(namespace, "connection_factory_namespace");
String connectionProducerName = config.getString(namespace,
"connection_producer_name");
String projectSearchUtilityNamespace = config.getString(namespace,
"project_search_utility_namespace");
String clientSearchUtilityNamespace = config.getString(namespace,
"client_search_utility_namespace");

//create the persistence instance through reflection
Class[] parameterTypes = null;
Object[] parameters = null;

if (connectionProducerName == null && projectSearchUtilityNamespace == null &&
clientSearchUtilityNamespace == null) {
   parameterTypes = new Class[] {String.class};
```

```
    parameters = new String[] {dbNamespace};
} else if (connectionProducerName != null && projectSearchUtilityNamespace == null &&
clientSearchUtilityNamespace == null) {
    parameterTypes = new Class[] {String.class, String.class};
    parameters = new String[] {dbNamespace, connectionProducerName};
} else if (connectionProducerName != null && projectSearchUtilityNamespace != null &&
clientSearchUtilityNamespace != null) {
    parameterTypes = new Class[] {String.class, String.class, String.class, String.class};
    parameters = new String[] {dbNamespace, connectionProducerName,
projectSearchUtilityNamespace, clientSearchUtilityNamespace};
} else {
    // throw a ConfigurationException here
}

persistence = (TimeTrackerProjectPersistence)
persistenceClass.getConstructor(parameterTypes).newInstance(parameters);
```

2) Id generation:

-obtain a Generator (Int32Generator instance):
  UUIDUtility.getGenerator(UUIDType.TYPEINT32)
-obtain an UUID (UUID32Implementation instance)
 UUID uuid = generator.getNextUUID().
 In this case an UUID32Implementation instance is obtained. UUID32Implementation
 represents the 32-bit UUID implementation of the UUID interface. Since 32 bits are used
there is no problem in obtaining an int from the uuid.
-obtain a String from the UUID
 String stringId = uuid.toString()
-obtain an id:
 int id = Long.valueOf(stringId.toString(), 16).intValue();

3) Common algorithm for atomic batch operations (except read operation)

```
// Start transaction
connection.setAutoCommit(false);

// Prepare a statement (SQL statements should be reused from version 1.0)
// Cache this statement in a map to avoid re-compilation every time it is used
PreparedStatement statement = connection.prepareStatement(sqlStatementString);

// Loop through all the entities to process
{
            // In the loop perform such operations:

            // Set SQL statement parameters for a particular entity,
            // via statement.setXXX() methods

            // After setting the parameters, execute the statement
            statement.executeUpdate();
}

// Commit transaction (if all operations were successful)
connection.commit();

// If any error happens, the transaction should be rolled-back
```

```
connection.rollback();

// Turn-off transactions
connection.setAutoCommit(true);
```

4) Common algorithm for non-atomic batch operations (except read operation)

```
// Turn-off transactions
connection.setAutoCommit(true);

// Prepare a statement (SQL statements should be reused from version 1.0)
// Cache this statement in a map to avoid re-compilation every time it is used
PreparedStatement statement = connection.prepareStatement(sqlStatementString);

// Loop through all the entities to process
{
        // In the loop perform such operations:

        // Set SQL statement parameters for a particular entity,
        // via statement.setXXX() methods

        // After setting the parameters, execute the statement
        statement.executeUpdate();
}

// Note, that the exceptions caught during the processing
// should not break the loop.
// Instead of this, they should be stored in the Throwable[] array.
// The number of elements in array should correspond to the number of the entities.
// When a particular entity is successfully processed,
// the array will contain null in the corresponding place.
// When any error happens, the array will contain
// cause exception in the corresponding place.
```

5) Common algorithm for the batch read operations

```
//  Prepare a statement (SQL statements should be reused from version 1.0)
// Cache this statement in a map to avoid re-compilation every time it is used
PreparedStatement statement = connection.prepareStatement(sqlStatementString);

// Loop through all the entities ids to process
{
        // In the loop perform such operations:

        // Set SQL statement parameters for a particular entity to be retrieved,
        // via statement.setXXX() methods

        // After setting the parameters, execute the query
        ResultSet resultSet = statement.executeQuery();
        // The resulting ResultSet should be processed in the same way,
        // as in the version 1.0.
        // It will result in creating instance of a particular entity,
        // which will be put into the result array.
}

// Note, that if in non-atomic mode, the exceptions should not break the loop.
// The failure of retrieving individual entities will be indicated through null
// set in the result array, instead of a particular entity instance
// If none of the entities is successfully retrieved, then throw an exception.
```

6) Build search condition string from the search Filter instance.

This algorithm describes DatabaseSearchUtility.buildCondition() method.

```
if (searchFilter instanceof ValueFilter) {
    ValueFilter filter = (ValueFilter) searchFilter;

    // get the column name of the DB table from the alias
    String columnName = getColumnName(filter.getFieldName());

    // build the SQL query with compare operation
    buffer.append("(");
    buffer.append(columnName);
    buffer.append(" ");
    buffer.append(filter.getOperation().toString());
    buffer.append(" ?)");

    // add the parameter value to the list
    parameterValues.add(filter.getValue());
} else if (searchFilter instanceof MultiValueFilter) {
    MultiValueFilter filter = (MultiValueFilter) searchFilter;

    // get the column name of the DB table from the alias
    String columnName = getColumnName(filter.getFieldName());

    // build the SQL query with "IN" predicate
    buffer.append("(");
    buffer.append(columnName);
    buffer.append(" IN (");

    // add the parameter values to the list at the same time
    Object[] values = filter.getValues();

    for (int i = 0; i < values.length; i++) {
        buffer.append("?,");
        parameterValues.add(values[i]);
    }

    // remove the last comma
    buffer.setLength(buffer.length() - 1);
    buffer.append("))");
} else if (searchFilter instanceof BinaryOperationFilter) {
    BinaryOperationFilter filter = (BinaryOperationFilter) searchFilter;
    // build the SQL query with binary operation, the left and right operands
    // generate the SQL query and add parameter values recursively
    buffer.append("(");
    buildCondition(filter.getLeftOperand(), buffer, parameterValues);
    buffer.append(filter.getOperation().toString());
    buildCondition(filter.getRightOperand(), buffer, parameterValues);
    buffer.append(")");
} else if (searchFilter instanceof NotFilter) {
    NotFilter filter = (NotFilter) searchFilter;

    // build the SQL query with "NOT" operation, the operand
    // generates the SQL query and adds parameter value recursively
    buffer.append("(NOT");
    buildCondition(filter.getOperand(), buffer, parameterValues);
    buffer.append(")");
} else {
    throw new IllegalArgumentException("searchFilter is not supported");
}
```

1.4 Component Class Overview

Project
This class holds the information about project. When creating an instance of this class the user has two options:
1) Use the default constructor and allow the GUID Generator component to generate a unique id
2) Use one of the parameterized constructors and provide an id for the Project instance; if the id already is contained by another project from the Projects table, then the newly created project will not be added to the Projects table
Also the user should not populate the creationDate and modificationDate fields, because if he does, the entry will not be added to the database. This fields will be handled automatically by the component(the current date will be used). When loading from the persistence, all the fields will be properly populated.

The 2.0 version added a companyId field with setter and getter, and also new constructors for an alternative way of work with companyId.

Client
This class holds the information about a client. When creating an instance of this class the user has two options:
1) Use the default constructor and allow the GUID Generator component to generate a unique id
2) Use one of the parameterized constructors and provide an id for the Client instance; if the id already is contained by another client from the Clients table, then the newly created client will not be added to the Clients table.
Also the user should not populate the creationDate and modificationDate fields, because if he does, the client will not be added to the database. This fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.
The 2.0 version added the companyId field with setter and getter, and also new constructors for an alternative of how to work with companyId.

ProjectManager

This class holds the information about an project manager. It has the following fields: project, managerId, creationDate, creationUser, modificationDate, modificationUser. When creating an instance of this class the user has two options:
1) Use the default constructor which does nothing
2) Use the parameterized constructor and provide an id and a project argument. The user should not populate the creationDate and modificationDate fields, because if he does, the project manager will not be added to the database. This fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.

ProjectWorker
This class holds the information about an project worker. It has the following fields: project, workerId, startDate, endDate, payRate, creationDate, creationUser, modificationDate, modificationUser. When creating an instance of this class the user has two options:
1) Use the default constructor which does nothing
2) Use the parameterized constructor and provide an id and a project argument. The user should not populate the creationDate and modificationDate fields, because if he does, the project worker will not be added to the database. This fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.

ProjectPersistenceManager
This manager is responsible for reading the configuration file. To accomplish this it will use the Configuration Manager component. From the configuration file two properties can be read:
-a class name identifying an implementation of TimeTrackerProjectPersistence(this property is required)
-a connection producer name identifying a ConnectionProducer instance(this property is optional). This instance of ConnectionProducer will provide the connection to the database.
Using these two properties this manager will create thorough reflection an TimeTrackerProjectPersistence implementation instance which will be used by the other two managers. These managers will use the getPersistence method to access the TimeTrackerProjectPersistence.

ClientUtility
The ClientUtility is useful to manage the clients.
. It can do the following things:
-add a client(Client instance) to the Clients table; if the Client instance has the id=-1 this manager will use the GUID Generator to generate an id for the Client instance;
in 2.0 version are added validations for companyId for a client, and validation for client name.
-delete a client from the Clients table
-delete all the clients from the Clients table
-retrieve a client(given its id) from the Clients table
-retrieve all the clients from the Clients table
-update a client in the client table;
in 2.0 version are added validations for companyId for a client, and validation for client name.
-add a project to a client
in 2.0 version are added constraints for projects and clients to have the same companyId if are linked.
-retrieve a certain project(specified by its id) of a client

This manager will receive in the constructor an ProjectPersistenceManager instance. It will use this instance to gain access to the database. It will use the getPersistence method from the ProjectPersistenceManager instance to obtain the persistence

Since version 1.1 this class also supports the batch versions of the CRUD operations and the search functionality.

ProjectUtility
The ProjectUtility is useful to manage the projects.
. It can do the following things:
-add a Project(Project instance) to the Projects table; if the Project instance has the id=-1 this manager will use the GUID Generator to generate an id for the Project instance
-delete a project from the Projects table
-delete all the clients from the Projects table
-retrieve a project(given its id) from the Projects table
-retrieve all the projects from the Projects table
-update a project in the Projects table
-assign a client to a project
-retrieve the client of a specified project
-add/remove/update/retrieve workers
-assign/remove/retrieve project manager
-add/remove/retrieve time entries
-add/remove/retrieve expense entries
This manager will receive in the constructor an ProjectPersistenceManager instance. It will use this instance to gain access to the database. It will use the getPersistence method from the ProjectPersistenceManager instance to obtain the persistence

Since version 1.1 this class also supports the batch versions of the CRUD operations and the search functionality.
In 2.0 version was added validations for some of the methods.

TimeTrackerProjectPersistence
TimeTrackerProjectPersistence represents the interface for data access. Client can choose between alternative implementations to suit persistence migration. Interface defines all necessary methods to interact with the database. This release comes with an Informix implementation.The methods exposes by this interface are very raw (it would be hard for a user to use them to obtain the functionality). They are aimed to an efficient database implementation (using INSERT, SELECT, UPDATE and DELETE statements) but other storage technologies can be used just as well (such as XML).

Version 1.1 added such functionality:
- batch versions of client/project CRUD operations.
- client/project search functionality

Version 2.0 added possibility to get the companyId for a specific kind of entity. This was necessaries since in this way it is abstract what class will be used for Company or how the links are between custom components and Company entity.

InformixTimeTrackerProjectPersistence
This class is a concrete implementation of the TimeTrackerProjectPersistence interface that uses an Informix database as persistence. This implementation uses the DB Connection Factory component to obtain a connection to the database. Transaction should be employed to ensure atomicity. This class provides two constructors. If the default constructor is used to create an instance of this class, then the DEFAULT_CONNECTION_PRODUCER_NAME constant will be used to obtain a

connection from the DB Connection Factory component. If the parameterized constructor is used then the user has the possibility to specify a name that will be passed to the DB Connection Factory component to obtain a connection to the Informix database.

Version 1.1 added such functionality:
- batch versions of client/project CRUD operations.
- client/project search functionality

Version 2.0 added posibility to obtain companyId for a specific id for time tracker entities: (for client, project, user_account, time entry and expense entry).

DatabaseSearchUtility
This class is used to perform search operations, using the specified search Filter.
It is used by the InformixTimeTrackerPersistence class, but can safely be reused by other DB persistence implementations.
Main its function is to create PreparedStatement representing the Filter and initialize its parameters to proper values. This class is configured via ConfigManager.

Filter
This interface represents a search filter. A search filter is used to specify what entities to search. In this component it is used to specify what projects or clients to search. Note, that the filter itself acts just like a data-structure, which specifies the parameters of search; it doesn't contain any search logic.

ValueFilter
This class implements the Filter interface.
This class represents a search filter, which compares the values of the specified field of the entity with the specified value, using the specified CompareOperation.
The entity is accepted if the operation returns true value, and is not accepted otherwise.
The supported operations are:

- equal to
- less than
- less than or equal to
- greater than
- greater than or equal to
- like (compare a string filed with a specified pattern string)

CompareOperation
This class is derived from the Enum class.
This class represents an enumeration type, representing the compare operations to be used by the ValueFilter.
It has such values:

- EQUAL - "=" operation, the result filter accepts only the entities, the specified field of which is equal to the specified value
- LESS - "<" operation, the result filter accepts only the entities, the specified field of which is less than the specified value
- LESS_OR_EQUAL - "<=" operation, the result filter accepts only the entities, the specified field of which is less than or equal to the specified value
- GREATER - ">" operation, the result filter accepts only the entities, the specified field of which is greater than the specified value
- GREATER_OR_EQUAL - ">=" operation, the result filter accepts only the entities, the specified field of which is greater than or equal to the specified value
- LIKE - "like" operation, the result filter accepts only the entities, the specified field of which matches specified pattern.

MultiValueFilter

This class implements the Filter interface.
This class represents a search filter, which compares the values of the specified field of the entity with the specified array of values. The entity is accepted if field value is equal to one of the values in the array. This filter corresponds to the "IN" predicate in SQL query.

NotFilter
This class implements the Filter interface.
This class represents a search filter, "negating" a search filter. It performs logical "NOT" operation on the specified filter. It means, that this filter will accept only these values, which are not accepted by the specified filter.

BinaryOperationFilter
This class implements the Filter interface. This class represents a search filter, combining two other search filters. The specified BinaryOperation is used to combine the filters. It can be either logical "AND" or logical "OR" operation.

- "AND" operation means, that this filter accepts only the values,which are accepted by both of the specified filters.
- "OR" operation means, that this filter accepts the values,which are accepted at least by one of the specified filters.

BinaryOperation
This class is derived from the Enum class.
This class represents an enumeration type, representing the binary logical operations to be used by the BinaryOperationFilter.
It has such values:

- AND - logical "AND" operation, the result filter accepts only the entities, accepted by both operand filters
- OR - logical "OR" operation, the result filer accepts the entities, accepted by at least one of the operand filters

1.5 Component Exception Definitions

Persistence Exception[custom]
The PersistenceException exception is used to wrap any persistence implementation specific exception. These exceptions are thrown by the TimeTrackerProjectPersistence interface implementations. Since they are implementation specific, there needs to be a common way to report them to the user, and that is what this exception is used for.
This exception is originally thrown in the persistence implementations. The business logic layer (the manager classes) will forward them to the user.

ConfigurationException[custom]
This exception is thrown by the ProjectPersistenceManager if anything goes wrong in the process of loading the configuration file or if the information is missing or corrupt.
It is also thrown by the DatbaseSearchUtility class when any error happens, when loading the configuration parameters.

InsufficientDataException[custom]
This exception is thrown when some required fields (NOT NULL) are not set when creating or updating an entry, type or status in the persistence. This exception is thrown by the ProjectUtility and ClientUtility.
This exception it is also used to specify if companyId value is not correct when are made operations for entities.

NullPointerException
This exception is thrown in various methods where null value is not acceptable.
Refer to the documentation in Poseidon for more details.

IllegalArgumentException
This exception is thrown in various methods if the given string argument is empty. Refer to the documentation in Poseidon for more details.
This exception it is also used to specify that relations between entities lead to the same companyId value for those entities.

BatchOperationException[custom]
This exception is thrown by the implementations of TimeTrackerProjectPersistence interface, when the batch operation fails.
It is also rethrown by the ProjectUtility and ClientUtility classes.
It stores the array of cause exceptions, each corresponding to the particular operation.
If a particular operation was successfull the array contains null in that place.

1.6 Thread Safety

This component is not thread safe. Thread safety was not a requirement.
Conflicts may occur in the persistence layer. In order to avoid them one would have to synchronize all the methods from the persistence implementations, to avoid concurrent database access. In addition, before making any change to the database, one would have to make sure the existing data is still valid.

In order to achieve this goal the database operations have to be atomic. The solution is to use JDBC transactions. The Connection class provides the setAutoCommit method to switch into manual commit mode. After that, the SQL statement that need to be executed in one transaction are executed in sequence. At the end the commit or rollback methods are used depending on whether everything was successful or an error occurred. If commit is successful then we have the guarantee that everything was executed atomically.

The 1.1 version doesn't add anything really new to the question of thread-safety. The information about mutability of particular classes can be looked-up in the docs tab of Poseidon. All the classes, representing the search filters are immutable, i.e. thread-safe. The DatabaseSearchUtility class is mutable, i.e. not thread-safe.

Version 2.0 doesn't add anything regards thread safety.

2. Environment Requirements

2.1 Environment

☐ Development language: Java 1.4

☐ Compile target: Java 1.4

2.2 TopCoder Software Components

☐ **Configuration Manager 2.1.4** - used to retrieve the configured data. This component is used by getting its singleton instance with ConfigManager.getInstance(). Then the existsNamespace method should be used to determine whether the namespace is already loaded. If not, the add method is used to load the default configuration file. Finally, getString returns the values of the properties.

☐ **GUID Generator 1.0**is used to assign unique ids to records. This component has the advantage of not requiring persistent storage (such as ID Generator requires), making the component easier to use. A generator is obtained with UUIDUtility.getGenerator(UUIDType.TYPEINT32). Then using generator.getNextUUID().toString() ids are generated as needed.

- **Base Exception 1.0** is used as a base class for the all the custom exceptions defined in this component. The purpose of this component is to provide a consistent way to handle the cause exception for both JDK 1.3 and JDK 1.4.
- **DB Connection Factory 1.0** provides a simple but flexible framework allowing the clients to obtain the connections to a SQL database without providing any implementation details. This component is used to obtain a connection to an Informix database(in the current implementation of Expense Entry component).

2.3 Third Party Components

Informix JDBC. Driver for connection to database.

3. Installation and Configuration

3.1 Package Name

com.cronos.timetracker.project
com.cronos.timetracker.project.persistence
com.cronos.timetracker.project.searchfilters

3.2 Configuration Parameters

Parameter	***Description***	***Values***
persistence_class	Fully qualified class name of the TimeTrackerProjectPersistence implementation. **Required**.	InformixTimeTrackerProjectPersistence
connection_factory_namespace	The namespace of the DB Connection Factory configuration file. **Required**.	com.topcoder.db.connectionfactory
connection_producer_name	Identifies a ConnectionProducer which will be used to obtain a connection to a database. **Optional**	Informix_Connection_Producer
project_search_utility_namespace	Specifies the ConfigManager namespace to use for retrieving configuration of DatabaseSearchUtility used to search projects. **Optional**	com.cronos.timetracker.project.persistence.DatabaseSearchUtility.projects
client_search_utility_namespace	Specifies the ConfigManager namespace to use for retrieving configuration of DatabaseSearchUtility used to search clients. **Optional**	com.cronos.timetracker.project.persistence.DatabaseSearchUtility.clients

Here is an example of the configuration file (the configurations file has been changed because of modifications for table names and column names):

```
<?xml version="1.0"?>
<CMConfig>
  <Config name="com.cronos.timetracker.project">
```

```
    <!-- the class name of the persistence (required) -->
    <Property name="persistence_class">

<Value>com.cronos.timetracker.project.persistence.InformixTimeTrackerProjectPersistence</
Value>
    </Property>

    <!-- the namespace of the DB Connection Factory configuration file (required) -->
    <Property name="connection_factory_namespace">
      <Value>com.topcoder.db.connectionfactory</Value>
    </Property>

    <!-- the name identifying a ConnectionProducer instance (optional) -->
    <Property name="connection_producer_name">
      <Value>Informix_Connection_Producer</Value>
    </Property>

    <!-- the ConfigManager namespace of DatabaseSearchUtility used to search projects
(optional) -->
    <Property name="project_search_utility_namespace">

<Value>com.cronos.timetracker.project.persistence.DatabaseSearchUtility.projects</Value>
    </Property>

    <!-- the ConfigManager namespace of DatabaseSearchUtility used to search clients
(optional) -->
    <Property name="client_search_utility_namespace">

<Value>com.cronos.timetracker.project.persistence.DatabaseSearchUtility.clients</Value>
    </Property>

  </Config>

  <Config
name="com.cronos.timetracker.project.persistence.DatabaseSearchUtility.clients">

    <!-- The SQL query template, required -->
    <Property name="query_template">
      <Value>select distinct client.client_id from client where </Value>
    </Property>

    <!-- The aliases of the DB column names, required -->
    <Property name="column_aliases">
      <Property name="Company ID">
        <Value>client.company_id</Value>
      </Property>
      <Property name="Name">
        <Value>client.name</Value>
      </Property>
      <Property name="Creation User">
        <Value>client.creation_user</Value>
      </Property>
      <Property name="Modification User">
        <Value>client.modification_user</Value>
      </Property>
      <Property name="Creation Date">
        <Value>client.creation_date</Value>
      </Property>
      <Property name="Modification Date">
        <Value>client.modification_date</Value>
      </Property>
    </Property>

  </Config>

  <Config
name="com.cronos.timetracker.project.persistence.DatabaseSearchUtility.projects">

    <!-- The SQL query template, required -->
    <Property name="query_template">
      <Value>select distinct project.project_id from project where </Value>
    </Property>

    <!-- The aliases of the DB column names, required -->
    <Property name="column_aliases">
      <Property name="Company ID">
        <Value>project.company_id</Value>
      </Property>
      <Property name="Name">
        <Value>project.name</Value>
      </Property>
      <Property name="Creation User">
        <Value>project.creation_user</Value>
      </Property>
      <Property name="Modification User">
        <Value>project.modification_user</Value>
      </Property>
      <Property name="Creation Date">
        <Value>project.creation_date</Value>
      </Property>
      <Property name="Modification Date">
        <Value>project.modification_date</Value>
      </Property>
    </Property>

  </Config>

  <Config name="com.cronos.timetracker.project.simple">
    <Property name="persistence_class">
      <Value>com.cronos.timetracker.project.persistence.SimplePersistence</Value>
    </Property>
    <Property name="connection_factory_namespace">
      <Value>com.topcoder.db.connectionfactory</Value>
    </Property>
  </Config>

  <Config name="com.topcoder.db.connectionfactory">
    <Property name="connections">
        <Property name="default">
            <Value>Informix_Connection_Producer</Value>
        </Property>
        <Property name="Informix_Connection_Producer">
            <Property name="producer">

<Value>com.topcoder.db.connectionfactory.producers.JDBCConnectionProducer</Value>
            </Property>
            <Property name="parameters">
                <Property name="jdbc_driver">
                    <Value>com.informix.jdbc.IfxDriver</Value>
                </Property>
                <Property name="jdbc_url">
                    <Value>jdbc:informix-
sqli://127.0.0.1:1526/time_tracker_project:informixserver=devinformix10_shm</Value>
                </Property>
                <Property name="user">
                    <Value>informix</Value>
                </Property>
                <Property name="password">
                    <Value>informix</Value>
                </Property>
            </Property>
        </Property>
    </Property>
  </Config>
</CMConfig>
```

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- ☐ Extract the component distribution.
- ☐ Execute the DDL insert script (test_files/testdata/TimeTracker.sql) to create the tables and some predefined entries.
- ☐ Edit the database configuration (test_files/timetrackerproject.xml).
- ☐ Execute ‘ant test’ within the directory that the distribution was extracted to.
- ☐ Execute the DDL delete script (test_files/delete.sql) to clean up the tables.

Exists a large variety of test cases that are all configured for version 2.0 (with all table, column, constraints).

4.2 Required steps to use the component

Extract the component distribution.

4.3 Demo

The configuration file showed above will be used in demo.

1) Basic functionality, version 1.0

```
//create a ProjectPersistenceManager
ProjectPersistenceManager pm = new
        ProjectPersistenceManager(“com.topcoder.timetracker.project.PersistenceMana
        ger”);

//create a Project instance using the parameterized constructor
Project project = new Project(1);

//From version 2.0 to a project you have to assign a companyId for it, before it is send to
        ProjectUtility to be saved.
Project.setProjectId(1);

// The same has to happened with client entities.

//using the setters the fields should be initialized; it is not necessary to perform
//this task in this demo since it is a trivial one, and the demo will get very big

//create a ProjectUtility
ProjectUtility pu = new ProjectUtility(pm);

//add the project to the database
pu.addProject(project);

//create a ProjectManager instance
ProjectManager projectManager = new ProjectManager(project, 1);

//assign the projectManager to the project
pu.assignProjectManager(projectManager);

//change the description of the project
project.setDescription(“Renault project”);
```

```
//update the project
pu.updateProject(project);

//query the project manager of the project
ProjectManager pm1 = pu.getProjectManager(project.getId());

//create a ProjectWorker
//ProjectWorker worker1 = new ProjectWorker(project,2);

//add the worker to the project(in the database)
pu.addWorker(worker1);

//create a second ProjectWorker
//ProjectWorker worke2r = new ProjectWorker(project,3);

//add the worker to the project(in the database)
pu.addWorker(worker2);

//update a worker
double PayRate = 100;
worker2.setPayRate(payRate);
pu.updateWorker(worker2);

//get a worker from the project
ProjectWorker worker3 = pu.getWorker(2,1);

//get all the workers from the project
List l = pu.getWorkers(1);

//add an expense entry to the project
pu.addExpenseEntry(5,1,"John");

//get all the expense entries from a project
List expenses = pu.getExpenseEntries(1);

//add a time entry to the project
pu.addTimeEntry(4,1,"John");

//get all the time entries from a project
List times = pu.getTimeEntries(1);

//create a ClientUtility instance
ClientUtility cu = new ClientUtility(pm);

//create a client using the parameterized constructor
Client client = new Client(5);
//using setters assume that the fields are initialized

//add the client to the database
cu.addClient(client);

//update the client description
client.setDescription("bussinesmen");
```

```
//update the client (in the database)
cu.updateClient(client);

//add a project to the client
cu.addProjectToClient(client.getId(), project);

//retrieve the projects of a client
List proj = cu.getAllClientProjects(client.getId());

//create another project
Project project1 = new Project(2);
//assume that the fields are initialized using the setters

//add the project to the database
pu.addProject(project1);

//assign a client to project1
pu.assignClient(2,5,”Tim”);

//query the client of a project
Client client1 = pu.getProjectClient(project1.getId());

//remove a worker
pu.removeWorker(2,1);

//remove all workers from a project
pu.removeWorkers(1);

//remove a time entry from a project
pu.removeTimeEntry(4,1);

//remove an expense entry from a project
pu.removeExpenseEntry(5,1);


//remove the project manager from a project
pu.removeProjectManager(2,1);

//remove a client
cu.removeClient(5);

//remove all the clients
cu.removeAllClients();
```

2) Batch operations with projects, clients and workers, version 1.1

```
// Assume that “pu” variable stores valid ProjectUtility instance
// Assume that “cu” variable stores valid ClientUtility instance

// Add new projects to the database, atomic mode
pu.addProjects(projects, true);

// Remove projects from the database, non-atomic mode
pu.removeProjects(projectIds, false);
```

```
// Add new clients to the database, atomic mode
pu.addClients(clients, true);

// Remove clients from the database, non-atomic mode
pu.removeClients(clientIds, false);

// Add new workers to the database, atomic mode
pu.addWorkers(workers, true);

// Remove workers from the database, non-atomic mode
pu.removeWorkers(workerIds, false);

// Other operations are quite similar, and there is no need to state them all here
// Generally the contract is such, the array is passed instead of the single parameter
// and the “atomic” boolean parameter specifies if the operation should be atomic.
```

3) Search functionality, version 1.1

```
// create a search filter, filtering the values in the specified column
// (they should be equal to the given value)
Filter equalFilter = new ValueFilter(CompareOperation.EQUAL, "Creation User",
"creationUser");

// create a search filter, filtering the values in the specified column
// (they should match a given pattern)
Filter likeFilter = new ValueFilter(CompareOperation.LIKE, "Name", "%Some pattern%");

// create a search filter, filtering the values in the specified column
// (they should be one of the values)
Filter multiValueFilter = new MultiValueFilter("Name", new Object[] {"name1", "name2"});

// create a search filter combining two filters using "OR" operation
Filter orFilter = new BinaryOperationFilter(BinaryOperation.OR, equalFilter, likeFilter);

// create a search filter combining two filters using "AND" operation
Filter andFilter = new BinaryOperationFilter(BinaryOperation.AND, equalFilter, likeFilter);

// create a search filter negating the given filter
Filter notAndFilter = new NotFilter(andFilter);

// search for projects using the filters
Project[] projects;

projects = pu.searchProjects(equalFilter);
projects = pu.searchProjects(likeFilter);
projects = pu.searchProjects(multiValueFilter);
projects = pu.searchProjects(orFilter);
projects = pu.searchProjects(andFilter);
projects = pu.searchProjects(notAndFilter);

// search for clients using the filters
Client[] clients;

clients = cu.searchClients(equalFilter);
clients = cu.searchClients(likeFilter);
clients = cu.searchClients(multiValueFilter);
clients = cu.searchClients(orFilter);
clients = cu.searchClients(andFilter);
clients = cu.searchClients(notAndFilter);
```

5. Future Enhancements

More TimeTrackerProjectPersistence implementations to this component.