Time Tracker Contact 3.2 Component Specification

Note that all v3.2 updates are in blue and all additions are in red.

1. Design

The Time Tracker Contact custom component is part of the Time Tracker application. It provides an abstraction of contacts and addresses. Contacts and Addresses exist in Time Tracker as a many to any relationship to many other entities. This component handles the persistence and other business logic required by the application.

This design provides ContactManager and AddressManager which can be used to add, update and retrieve Contacts and Addresses.

The Contacts/Addresses can be searched with filter. Predefined filters can be created by the filter factory. The user can also created custom filters by the Filters provide by Search Builder component.

The action which will modify the database can be audited according to user's setting. And the audit will be rollback if the transaction is failed.

And the data source is pluggable, new data source can be easily added.

New features in version 3.2:

- Introduces a session bean to facilitate transactional control for each of Contact and Address. This was a major issue with the 3.1 version. As a result, the DAO implementation documentation as well as the relevant SDs is modified to remove all manual transaction control.
- The v3.1 ContactManager and AddressManager will become delegates to these beans. As such, the managers will be split into an interface with this name, and a delegate implementation for the EJB integration.
- Only local access to the bean is needed.
- Transactional control uses the "required" setting to guarantee that any action is always under such control.

DESIGN CONSIDERATIONS:

<u>EJBs and ConfigManager:</u>

The EJB specification stipulates that File I/O is not allowed during the execution of the bean. At first glance this might mean that the use of the Config Manager, and by extension Object Factory, DB Connection Factory, and ID generator, could not be allowed because it performs property retrieval and storing using files. However, the restriction is only placed on the bean's lifetime, not the ConfigManager's. Therefore, as long as the Config Manager does not perform I/O itself during the execution of the bean, or to be more accurately, that the thread performing a bean operation does not cause the ConfigManager to perform I/O in the course of the bean's execution, the use of Config Manager to hold an in-memory library of properties is acceptable. Therefore, this design makes extensive use of Config Manager, Object Factory, DB Connection Factory, and ID generator, again, with the stipulation that the Config Manager implementation does not perform I/O when this component is retrieving properties. The current default implementation of the Config Manager holds all loaded properties in a map, so all property reading is done in-memory.

In general, and aside to the central issue here, the purpose behind the EJB specification stipulation against I/O during the business operation of the bean exists to accommodate portability. In practice, since portability is not always a reason behind the choice to use EJBs, many containers allow direct I/O in the bean. In fact, since the main purpose of the use of EJBs in this application is to enforce transactional control, I/O may be allowed as long as the access can be transitionally controlled, something that is a challenge in itself.

Serialization and Filter:

The Filter hierarchy in Search Builder is currently not serializable, which means it currently cannot be used. The PM is currently researching solutions to this.

JNDI and DBConnectionFactory:

The TxDataSource required for this component can still be obtained as a javax.sql.DataSource. As such, the JNDIConnectionProvider available in the DBConnection Factory component can still perform connection creation services for this design. As such, it is kept unaltered in the DAO implementations.

1.1 Design Patterns

- Data Access Object Pattern is used by InformixContactDAO & InformixAddressDAO.
- Strategy Pattern is used to allow plugging different types of data source.
- Factory Pattern is used to create different filters by ContactFilterFactory & AddressFilterFactory.
- Business Delegate Pattern is used by manager implementation so the user is decoupled from the intricacies of obtaining and calling the session EJBs.
- Facade Pattern is used by ContactManager & AddressManager.

1.2 Industry Standards

JavaBean
SQL
JDBC
EJB 2.1

1.3 Required Algorithms

The algorithms of Contact and Address are almost the same, therefore only Contact algorithms are listed below.

Note: For v3.2, all rollbacks occur in the EJB. Therefore, even the audit manager is not called programmatically to rollback the created audit.

1.3.1 Alias of column name

The column names used in filter factories are aliases. The SearchBundle will convert the aliases to the corresponding actual names defined in the SearchBundle configuration files. Every alias used by the filter factories should be defined in the “alias” property of the SearchBundle configuration files.

1.3.2 Call ContactBean ejbCreate

1. Obtain environmental values from JNDI for the namespace and DAO key.
2. Create the ObjectFactory with a new ConfigManagerSpecificationFactory created with the retrieved namespace
3. Create dao by the ObjectFactory with the retrieved key ContactDAOKey

1.3.3 Construct InformixContactDAO with given namespace

1. Get the configManager by ConfigManager.getInstance()
2. Get the connectionFactoryNamespace from the configManager with namespace and key as "connection_factory_namespace"
3. Create the connectionFactory by new DBConnectionFactoryImpl(connection_factory_namespace)
4. Get the connectionName from the configManager with namespace and key as "connection_name"
5. Get the idGeneratorName from the configManager with namespace and key as "idgenerator_name"
6. Get contactIDGenerator by IDGeneratorFactory.getIDGenerator(idGeneratorName)
7. Get the searchBundleName from the configManager with namespace and key as "search_bundle_name"
8. Get the searchBundleNamespace from the configManager with namespace and key as "search_bundle_namespace"
9. Create search bundle manager with searchBundleNamespace
10. Get searchBundle by manager.getSearchBundle(search_bundle_name)
11. If auditManager is null, get the auditManagerNamespace from the configManager with namespace and key as "audit_manager_namespace". Create the ObjectFactory with a new ConfigManagerSpecificationFactory created with the auditManagerNamespace. Create auditManager by the ObjectFactory with the key as "AuditManager"

1.3.4 Add contact

```
1. Throw InvalidPropertyException if any property of the contact is
   null.
2. Set the ID of the contact by idGenerator.getNextID()
3. Set the creation and modification date to current date.
4. If doAudit is true, create an auditHeader with the info of this
   contact and create it by AuditManager
5. Get a connection by createConnection, and create a prepared
   statement to add the contact to the contact table.
6. Set all the parameters of the statement according to all the
   properties of contact and execute the statement.
```

1.3.5 Add contacts

1. For each contact ...
   a) Throw InvalidPropertyException if any property of the contact is null.
   b) Set the ID of the contact by contactIDGenerator.getNextID()
   c) If doAudit is true, create an auditHeader with the info of this contact and create it by AuditManager
   d) Add the id of the header to the auditId which is created by new long[contacts.length]

... end each

2. Get a connection by createConnection and create a prepared statement to add the contact to the contact table.
3. For each contact, set all the parameters of the statement according to all the properties of contact and addBatch
4. Execute the batch, and get the returned int[]
5. If BatchUpdateException is thrown
   a) Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED, create BatchOperationException with the boolean array.
   b) Throw the BatchOperationException.

1.3.6 Update contact

1. Throw InvalidPropertyException if the id of the contact <=0, or any property of the contact is null.
2. If the bean is not changed(contact.isChanged() is false), simply return.
3. Set the modification date to current date.
4. If doAudit is true, get the contact with the id by retrieveContact. Create an auditHeader with the different and create it by AuditManager
5. Get a connection by createConnection, and create a prepared statement to update the contact.
6. Set all the parameters of the statement according to all the properties of contact and execute the statement.

1.3.7 Update contacts

1. Get all the ids of contacts
2. Get the orgContacts by retrieveContacts(ids)
3. For each contact ...
   a) Throw InvalidPropertyException if the id of the contact <=0, or any property of the contact is null.
   b) Set the modification date to current date.
   c) If doAudit is true, create an auditHeader with the difference between orgContact and contact, and create it by AuditManager
   d) Add the id of the header to the auditId which is created by new long[contacts.length]
4. ... end each
5. Get a connection by createConnection and create a prepared statement to update the contact in the contact table.
6. For each contact, set all the parameters of the statement according to all the properties of contact and addBatch
7. Execute the batch, and get the returned int[]
8. If the BatchUpdateOperation is thrown
   a) Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED
   b) Throw the BatchOperationException.

1.3.8 Remove contact

1. If doAudit is true, create an auditHeader with the info of this contact and create it by AuditManager
2. Get a connection by createConnection, and create a prepared statement to remove the contact from the contact table.
3. Set the parameter of the statement to the id of contact and execute it.

1.3.9 Remove contacts

1. Get the contacts by retrieveContacts(ids)
2. For each contact ...
   a) If doAudit is true, create an auditHeader with the info of this contact and create it by AuditManager
   b) Add the id of the header to the auditId which is created by new long[contacts.length]
3. ... end each
4. Get a connection by createConnection and create a prepared statement to remove the contact with given id from the contact table.
5. For each contact, set all the parameter of the statement to id of contact and addBatch
6. Execute the batch, and get the returned int[]
7. If the BatchUpdateOperation is thrown
   a) Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED, create BatchOperationException with the boolean array.
   b) Throw the BatchOperationException.

1.3.10 *Retrieve contact*

1. Get a connection by createConnection, and create a prepared statement to retieve the contact from the contact table with the contact type from contact_type table.
2. Set the id of contact in the statement and execute it.
3. If the record is found, create Contact with the ContactType (the id of the ContactType should equal the contact_id) according this record.
4. Else return null
5. Close the connection before return

1.3.11 *Retrieve contacts*

1. Get a connection by createConnection, and create a prepared statement to retieve the all the contacts with given ids from the contact table with the contact type from contact_type table.
2. Execute the statement.
3. For each record, create Contact with the ContactType (the id of the ContactType should equal the contact_id) according this record.
4. Close the connection and return an array containing all the Contacts

1.3.12 *Get all contacts*

1. Get a connection by createConnection, and create a statement.
2. Build sql query to retrieve contacts with given ids from the contact table with the contact type from contact_type table and execute it.
3. For each record, create a new Contact and ContactType (the id of the ContactType should equal the contact_id) according to the record
4. Close the connection and return an array containing all the Contacts

Note that getting all countries and states is very similar (details in the method documentation is provided)

1.3.13 *search contacts*

1. Call searchBundle.search(filter)
2. For each record
3. Create a new Contact and ContactType (the id of the ContactType should equal the contact_id) according to the record
4. ... end each
5. Return an array containing all the Contacts

1.3.14 *Associate contact with entity*

1. If doAudit is true, create an auditHeader with the info of this association and create it by AuditManager
2. Get a connection by createConnection, and create a prepared statement to add the association to the contact_relation table.
3. Set all the parameters of the statement according to the contact ID, contact type ID, entity ID and execute the statement.
4. Execute the statement.

1.3.15 *Deassociate contact with entity*

1. If doAudit is true, create an auditHeader with the info of this association and create it by AuditManager
2. Get a connection by createConnection, and create a prepared statement to delete the association to the contact_relation table.
3. Set all the parameters of the statement according to the contact ID, contact type ID, entity ID and execute the statement.

1.4 Component Class Overview

ContactManager:

This interface defines the contract for the complete management of a contact. It provides single and batch CRUD operations as well as the means to associate or disassociate a contact from an entity. It also provides the ability to audit each operation, if so desired. It has one implementation in this design: ContactManagerLocalDelegate.

Contact:

This class holds the information of a contact.

ContactDAO:

This interface specifies the contract for implementations of a Contact DAOs. A ContactDAO is responsible for accessing the database.

InformixContactDAO:

This class is the Informix database implementation of the ContactDAO. It provides general retrieve/update/remove/add functionality to access the database. And it provides SearchContact method to search contact with filter.

ContactType:

This enumeration represents the contact type.

ContactFilterFactory:

This class is used to create pre-defined filters or AND/OR/NOT filters which are used to search contacts. The column names used in this factory are aliases. The SearchBundle will convert the aliases to the actual names defined in the SearchBundle configuration files.

ContactHomeLocal:

The local interface of the Contact EJB

ContactLocal:

The local component interface of the Contact EJB, which provides access to the persistent store for contacts managed by the application.

ContactManagerLocalDelegate:

Implements the ContactManager interface to provide management of the Contact objects through the use of a local session EBJ. It will obtain the handle to the bean's local interface and will simply delegate all calls to it. It implements all methods.

ContactBean:

The session EJB that handles the actual manager requests. It simply delegates all operations to the ContactDAO it obtains from the ObjectFactory.

AddressManager:

This interface defines the contract for the complete management of an address. It provides single and batch CRUD operations as well as the means to associate or disassociate an address from an entity. It also provides the ability to audit each operation, if so desired. It has one implementation in this design: AddressManagerLocalDelegate.

Address:

This class holds the information of an address.

AddressDAO:

This interface specifies the contract for implementations of an Address DAOs. An AddressDAO is responsible for accessing the database.

InformixAddressDAO:

This class is the Informix database implementation of the AddressDAO. It provides general retrieve/update/remove/add functionality to access the database. And it provides SearchAddresses method to search addresses with filter.

AddressType:

This enumeration represents the address type.

AddressFilterFactory:

This class is used to create pre-defined filters or AND/OR/NOT filters which are used to search addresses. The column names used in this factory are aliases. The SearchBundle will convert the aliases to the actual names defined in the SearchBundle configuration files.

Country:

This class holds the information of a country.

State:

This class holds the information of a state.

AddressHomeLocal:

The local interface of the Address EJB

AddressLocal:

The local component interface of the Address EJB, which provides access to the persistent store for addresses managed by the application.

AddressManagerLocalDelegate:

Implements the AddressManager interface to provide management of the Address objects through the use of a local session EBJ. It will obtain the handle to the bean’s local interface and will simply delegate all calls to it. It implements all methods.

AddressBean:

The session EJB that handles the actual manager requests. It simply delegates all operations to the AddressDAO it obtains from the ObjectFactory.

1.5 Usage of Exceptions

ContactException:

This exception will be the base exception for all exceptions thrown by the ContactManager/AddressManager classes. This exception can be used by the application to simply their exception processing by catching a single exception regardless of the actual subclass.

IDGenerationException:

This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes when ID can't be generated successfully. This exception will be exposed to the caller of addContact/Address method.

InvalidPropertyException:
This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes when the property of the properties of given Contact is invalid. This exception will be exposed to the caller of addContact/Address(s) and updateContact/Address(s) method.

AuditException:
This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes if they encounter any exception when try to audit. This exception will be exposed to the caller of addContact/Address(s), updateContact/Address(s) and removeContact/Address(s) methods.

PersistenceException:
This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes when they encounter database exceptions. This exception will be exposed to the caller of database related methods.

ConfigurationException:
This exception will be thrown by the ContactManager/AddressManager and ContactDAO/AddressDAO implementations when they encounter a configuration error. This exception will be exposed to the caller of constructor of ContactManager/AddressManager and ContactDAO/AddressDAO implementations.

BatchOperationException:
This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes when any operation in the batch is failed. This exception will be exposed to the caller of batch operation methods.

EntityNotFoundException:
This exception will be thrown by the Contact/AddressManager classes, EJB entities, and Contact/AddressDAO classes when trying to update the Contact/Address which can be not found. This exception will be exposed to the caller of updateContact/Address(s) updateContact/Address(s) methods.

1.6 Thread Safety

Contact, Address, Country and State are not thread-safe by being mutable. They are not supposed to be used in multi-thread environment. If they would be used in multi-thread environment, they should be synchronized externally.

The classes in this component are thread-safe by being immutable and by the fact they are used in an EJB container, which assures that only one thread will use the session bean, and thus the DAO.

2. Environment Requirements

2.1 Environment

1 Development language: Java 1.4
2 Compile target: Java 1.4, Java 1.5

2.2 TopCoder Software Components

- Configuration Manager 2.1.5 is used to provide configuration options
- Object Factory 2.0 is utilized to create the implementations of Contact/AddressDAO.
- Base Exception 1.0 is used to provide a base for all custom exceptions.

- ID Generator 3.0 is used to generate IDs for Address and Contact.
- DB Connection Factory 1.0 is used to generate the database Connection.
- Type Safe Enum 1.0 is used to be extended by the ContactType and AddressType enumeration.
- Search Builder 1.3.1 is used to enable search with filter
- Time Tracker Audit 3.1 is used to provide audit functionality
- Time Tracker Common 3.1 provides TimeTrakerBean base class.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.contact

com.topcoder.timetracker.contact.ejb

com.topcoder.timetracker.contact.persistence

3.2 Configuration Parameters

3.2.1 ContactManagerLocalDelegate

Property Name	Description	Details	Required
jndi_reference	JNDI reference to the local contact EJB.	Example: “java:comp/env/ejb/ ContactLocal”	Yes

3.2.2 AddressManagerLocalDelegate

Property Name	Description	Details	Required
jndi_reference	JNDI reference to the local address EJB. Required	Example: “java:comp/env/ejb/ AddressLocal”	Yes

3.2.3 InformixContactDAO

The InformixContactDAO has the following configuration using the specified namespaces (or one provided by the application or ContactManager):

Namespace: com.topcoder.timetracker.contact.persistence.InformixContactDAO

Property Name	Description	Details	Required
connection_factory_namespace	The namespace used to create DBConnectionFactory	String	Yes
connection_name	The name used to get connection from connection factory	String	No
idgenerator_name	The name used to get the IDGenerator	String	Yes
search_bundle_name	The name will be used to get SearchBundle from SearchBundleManager	String	Yes

search_bundle_namespace	The namespace used to create `SearchBundleManager`	String	Yes

NOTE: all the aliases used in the ContactFilterFactory should be defined in the alias property of the configuration file.

3.2.4 InformixAddressDAO

The InformixAddressDAO has the following configuration using the specified namespaces (or one provided by the application or AddressManager):

Namespace: com.topcoder.timetracker.address.persistence.InformixAddressDAO

Property Name	**Description**	**Details**	**Required**
connection_factory_namespace	The namespace used to create DBConnectionFactory	String	Yes
connection_name	The name used to get connection from connection factory	String	No
idgenerator_name	The name used to get the IDGenerator	String	Yes
search_bundle_name	The name will be used to get SearchBundle from SearchBundleManager	String	Yes
search_bundle_namespace	The namespace used to create `SearchBundleManager`	String	Yes

NOTE: all the aliases used by the AddressFilterFactory should be defined in the alias property of the configuration file.

3.2.5 Sample Configurations

Please refer to contact_config.xml and address_config.xml for J2SE configuration. For J2EE configuration, please refer to contact_j2ee_config.xml, address_j2ee_config.xml, and the sample deployment descriptor ejb-jar.xml. All are in the docs directory.

3.3 Dependencies Configuration

The dependency components should be configured according to their documentation.

The EJB deployment descriptor must have a data source configured for the DBConnection Factory. Also, it must include the following two environment entries for the CompanyBean:

Parameter	**Description**	**Details**
SpecificationNamespace	Namespace to use with the ConfigManagerSpecificationFactory Required	Example: “com.topcoder.specification”
ContactDAOKey AddressDAOKey	Key to the CompanyDAO/ AddressDAO instance to pass to object factory Required.	“companyDAOKey” “addressDAOKey”

The sample deployment descriptor introduced in 3.2.5 has a simple example of these entries being used.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ant test

4.2 Required steps to use the component

Nothing special required

4.3 Demo

Demo of Address is same with contact, except for the retrieval of all states and countries. This will be demoed in 4.3.5.

For the purposes of this demo, the configuration introduced in 3.2.5 can be used.

4.3.1 Create ContactManagerLocalDelegate

```
//Create a new ContactManagerLocalDelegate, the default namespace will be used
ContactManagerLocalDelegate manager = new ContactManagerLocalDelegate();

// Create a new ContactManagerLocalDelegate with namespace
ContactManagerLocalDelegate manager = new
ContactManagerLocalDelegate( “com.topcoder.timetracker.contact.ContactManagerLocalDelegate”);
```

4.3.2 Manager Contacts

```
//Add the contacts by contact manager, contacts is Contact[], and this action will be audited
manager.addContacts(contacts, true);

//get all contacts by manager
Contact[] contacts = manager.getAllContacts();

//remove one contact, this action will be audited
manager.removeContact(Contact[1].getId(), true);

//change the country of one
contacts[3].setCountry(country_us)

//Update the contact, this action will be audited
manager.updateContact(contacts[3], true);
```

4.3.3 Search Contacts

```
//Create filter with country US
Filter myFilter = ContactFilterFactory.createCountryFilter(country_us);

//search with this filter, the result will include the contacts whose country is US
Contact[] contacts = manager.searchContacts(myFilter);
```

4.3.4 Manage Contact Relation

```
//Add the contact when adding a client, a client is come from Time Tracker Client component, it
```

```
contains contact and address, this action won't be audited
manager.addContact(client.getContact(), false);

//Associate the contact with this client, this action won't be audited
manager.associate(client.getContact(), client.getId(), false);


//Deassociate the contact with the client, this action won't be audited
manager.deassociate(client.getContact(), client.getId(), false);

//remove the contact when removing a client, this action won't be audited
manager.removeContact(client.getContact().getId(), false);
```

4.3.5 Retrieve State and Country data

```
//Assume that an AddressManager instance is created in the same manner as the ContactManager
//in 4.3.1. Use it for the following actions.

// Get all states
State[] states = addressManager.getAllStates();

// Get all countries
Country[] countries = addressManager.getAllCountries();
```

5. Future Enhancements

Implement accessibility for other database systems.