Search Builder 1.3 Component Specification

1. Design

The Search Builder component provides an API for both simple and complex searches against a persistent data store (i.e. LDAP, Database). The present version supports both database searches and LDAP searches. The component allows a user to programmatically create searches. It will translate them into the appropriate query string, execute the query against the configured data store connection, and return the result set to use for further processing.

In addition, the component is designed so that future sub-components can be plugged in to provide searches of additional types of persistent data stores. For example, in the future someone might need to provide a file system search.

Backwards Compatibility

Note that the documentation in this section refers to the original v1.1 design with regards to the v1.0 design.

Since the original v1.0 design is tightly coupled with database, it is impossible to keep original class hierarchy and interfaces. Since the project manager allows a redesign (see development forum), this design re-factored the old version. Here are the major changes made in the new version:

- Use external package com.topcoder.db.connectionfactory to handle database connections
- Use external package com.topcoder.util.datavalidator to handle data validation
- Re-factor filter package to decouple build query string function from filter classes. Thereby, the filter can be built once and converted to filter strings for different data stores.
- Remove all database specific interfaces in SearchBundle class and add a set of Builder() methods to provide a set of high level, simplified interfaces for this component. Thus clients do not need to be aware of class hierarchy in the filter package, simplifying the use of this component.

Besides these major changes, some minor changes are made to support both LDAP and database. However, all functions from the old version for database are kept, including supporting table names and alias names. For database connection settings, functions are kept by using DBConnectionFactory component. For validation settings, functions are kept by using DataValidation component.

This design provides the following required functionalities:

- It allows user applications to build search filter programmatically, including both associative filters and simple filters.

- It allows user application to build a Search Bundle Manager to manage multiple search bundles.

- It provides APIs to build a search bundle. In addition, it allows user application to set connection information bound to the search bundle. Thus the data store bound to the search bundle is pluggable.

- This component provides interfaces to allow user applications to execute

configured search bundles.

- This component provides two sub-components to support database searches and LDAP searches, respectively. Database sub-component also has the ability to execute prepared statements.

- This component utilizes the external package com.topcoder.util.datavalidator to build validation rules for each searchable field.

This design provides the following additional functionalities:

- It allows user application to build search bundle manager through a reading configuration file.

- It provides APIs in the Search Bundle Class to allow user application to build each individual filter without knowing the underlying hierarchy of filter classes

- The component supports alias names.

- The component also makes searchable fields containing validation rules pluggable, in addition to connection information.

This version aims to fix two issues with the previous implementation, with no functional additions. Some minor issues that did not deal directly with the functionality of the component have also been fixed.

Search Builder v1.3 Design Changes

Required Changes

- The component now closes its connections to the LDAP server and the database after executing the search.
- A SearchStrategy interface has been added. This is the layer from which the necessary steps are taken to translate the Filters into a format understandable by the relevant persistent store. Parameter binding on the PreparedStatement is done for the Database Sub-component.
- A NullFilter has been added to the filters subpackage. This is used to constrain searches to fields that are null.

Optional Changes

- The SearchFilter constants and related method getFilterType have been deprecated. Reliance on these constants hinder the extensibility of the component, because the interface needs to be updated everytime a new Filter was needed.
- On a related note, the buildXXX methods in SearchBundle have also been deprecated for the same reasons. The buildXXX methods added minor convenience to the client (the method signature was very similar to the constructor signatures of the Filter classes), and caused a scope violation by making the SearchBundle responsible for creating Filters as well as managing the context and aliases.
- The ConnectionStrategy, SearchStringBuilder and ConnectionInfo classes have been removed. They provided assumptions on the underlying persistent store which may not necessarily be true. Instead, the persistent store is abstracted into a much simpler SearchStrategy interface. This hides any details of the persistent store from the bundle, and simply exposes a search method.
- As a side effect the SearchBundle is no longer responsible for controlling the

interaction between the SearchStringBuilder and ConnectionStrategy. Instead, it takes on the more reasonably scoped role of maintaining the search context, validation and aliasMaps.

- Building the Search String has now been broken down. Several classes are now responsible for building the String based on the relevant Filter. In the previous version, all the search String-building steps were placed in a single class. Supporting new Filters meant having to recompile that class in order to support it. In the new version, a ClassAssociator (from ClassAssociation component) is used to relate Filters to their respective SearchFragmentBuilder. Supporting a new Filter simply means writing up a new handler and plugging it into the class associator.
- The SearchableFields is handled more cleanly. This is accomplished in two steps: The first step addresses the fact that the old configuration did not allow Object Validators to be specified for a searchable field. Configuration has been modified to support it. The second step allows for no Object Validator to be specified for a searchable field. In the previous version, if a field had no validator, it was immediately assumed to be invalid (and rejected by the SearchBundle). In the new version, a field can now be specified to be searchable without constraints, instead of forcing the user to specify an Object Validator when validation was not desired.

1.1 Industry Standards

JDBC 2.0
JAVA 1.4
SQL92 Standard
JNDI

1.2 Design Patterns

- Strategy – The involved packages are com.topcoder.search.builder, com.topcoder.search.builder.ldap, and com.topcoder.search.builder.database. The involved classes are: SearchStrategy and SearchFragmentBuilder classes. They provide a pluggable framework for executing configured search for different data stores.

- Template Method Pattern – The involved class are LDAPSearchStrategy and DBSearchStrategy abstract class and its implementers. The algorithms of search() method is defined, which calls buildSearchContext() method and connect() method (for LDAP). Subclasses can implement those two methods differently to meet their purpose.

- State Pattern - The SearchContext represents the state of building up an object as it is delegated from each fragment builder.

- Builder Pattern - The concrete DBSearchStrategy and LDAPSearchStrategy build up the Search String based on the structured Filter object that is provided.

- Composite Pattern – The involved class are the classes in the com.topcoder.search.builder.filter package. The Filter interface is a common interface for both individual and composite components so that clients can view both of them uniformly. This is useful in building the tree structure of composite filters.

1.3 Required Algorithms

The classes provided by the component are relatively easy and do not implement very complex logic. Most algorithms are explained in the form of sample code or pseudo code in the documentation tab of each class method. Algorithms need to be further explained are listed as follows:

- Validation Algorithm – The validation algorithm is different for each individual filter. For composite filters, the key point is to verify if each component filter has a searchable field and if the value of field is valid according to the rule. It does not verify And/Or logic. The complete algorithms are listed in the documentation tab for each concrete filter class. The following algorithm can be used to validate the OrFilter:

```
//my code to ckeck isValid
if ((validators == null) || (alias == null)) {
            throw new NullPointerException(
                "The param should not be null to check Valid");
        }

        for (int i = 0; i < filters.size(); i++) {
            Filter tmp = (Filter) filters.get(i);

            ValidationResult v = tmp.isValid(validators, alias);

            //success then continue
            if (v.isValid()) {
                continue;
            }

            //if fail the return the fail result
            return ValidationResult.createInvalidResult("fails",
                v.getFailedFilter());
        }

        //valid success
        return ValidationResult.createValidResult();
```

- Build Search String From LikeFilter:
  The LikeFilter is designed to support both substring and wildcard searching. It can be implemented since the underlying data stores (Database and LDAP) support wildcard searching. The implementation of the algorithm is described in the detail in the documentation tab of DatabaseSearchStringBuilder.buildFromLikeFilter method and LDAPSearchStringBuilder.buildFromLikeFilter method, respectively. The following key points describe how the algorithm works:

  1) Substring Searching: The searching value must start with “SS:”, “SW:” or “EW:”. They represent “Contains”, “Starts With” and “Ends With” respectively. When generate the search string, we add the wildcard according the search type. For example, if the value is “SS:abc”, the search string for database is “%abc%” and for LDAP is “*abc*”.

     When using this searching type, the user/application doesn’t need to worry about the escape of wildcard.

  2) Wildcard Searching: The users/applications use the wildcard by themselves. It’s more powerful. It starts with “WC:”

For example: it can search the string like “%abc%ABC%” or “*abc*ABC*”.

- Revised DB Search Algorithm:
  The revised database search is broken down in the multiple Database SearchFragmentBuilders. Each SearchFragmentBuilder is responsible for generating an SQL fragment that contains a ‘?’ as a bindable JDBC parameter. The SearchFragmentBuilder appends this SQL fragment to the SearchContext, and then adds the actual Filter as a BindableParameter. The details of the SQL fragment to build for each specific filter is specified within the method documentation tab in the ZUML file.

  Once all the filters have been evaluated by the SearchFragmentBuilder, the DBSearchStrategy will bind the bindable parameters to a PreparedStatement that has been generated from the built SQL Strings.

  See the Sequence Diagrams *Search Builder v1.3 Build Search Fragment on Context Sequence Diagram* and *(v1.3) Search Builder Execute Built DB Search Sequence Diagram* for more details.

1.4 Component Class Overview

Below is a brief overview of the classes in this component. Please refer to the class diagram’s documentation tab for a more complete overview of each class. For the external classes used in the component, such as ConfigManager class, please consult the related documentation for a more detailed and comprehensive description.

Package com.topcoder.search.builder

SearchStringBuilder (interface):
This interface only defines one method. The method is used to construct the search string from the filter. Each data store sub-package should implement this interface to convert the filter to a specific search string.
This class has been deprecated and been replaced by SearchFragmentBuilder.

ConnectionInformation (interface):
This interface includes all the information needed to connect to a data store, such as LDAP and database. It defines APIs to retrieve the information.
This class has been deprecated.

ConnectionStrategy (abstract class):
The abstract class specifies the APIs for searching data store, such as LDAP and databases. In order to search a data store, there must be a corresponding concrete implementation of this class. The design provides two concrete implementations to work with LDAP and Database, respectively.
This class has been deprecated and replaced by SearchStrategy.

SearchBundleManager (class):
This class manages a collection of SearchBundle objects. It provides APIs to allow user application to manage the collection of SearchBundle objects.
This class has been modified to use the new configuration based on the changes to the design.

ValidationResult (class):

This class models the filter validation result. It is used by SearchBundle class and all classes in the com.topcoder.search.builder.filter package. It includes the following information: is the filter valid? (boolean); a message and a filter object, if validation fails. It provides APIs to retrieve the information.

SearchBundle (class):
This class represents a search bundle. It is the most important class of the component. It provides APIs to allow user dynamically set connection information and validation rule. It also allows clients to build filters and execute searches.
This class has been modified to use the new interfaces like SearchStrategy.

SearchContext (class):
This is a State object that is used to hold the current status of building a Search. It contains any bindable parameters, as well as the existing search string. It may be used by any SearchStrategies as a convenience class to help with building the search string and relevant parameters.

SearchFragmentBuilder (interface):
This is the interface that is used to define the contract for building a SearchContext out of a provided filter. Implementations usually come as a 'family' to support a specific type of datastore. Each member of the family will correspond to one or more of the Search Builder Filters. Adding filters may entail adding additional SearchFragmentBuilders, and adding a new type of datastore may require creating a new family of SearchFragmentBuilders.

SearchStrategy(interface):
The search strategy is responsible for connecting to the datastore, building the necessary query String (if applicable) and executing it against the datastore. The object returned from the search is dependent on the type of datastore that is used.

AlwaysTrueValidator(package-private class):
This validator is always true for any provided value. It is used as a substitute value for null constraints on searchable field maps. This is done as a convenient alternative to encapsulating null-handling behavior into all filter classes.

Package com.topcoder.search.builder.filter

All Existing Filter classes in this package have been modified in the following manner:

- The clone method should be revised to follow the correct method of cloning (using super.clone())
- The getFilterType methods (and corresponding constants) have been deprecated.

Filter (interface):
This interface defines APIs that every implementation must adhere to. It is part of the composite pattern. It is a common interface for both individual and composite components so that both the individual components and composite components can be viewed uniformly.

AbstractSimpleFilter (abstract class):
This abstract class is a concrete implementer of the Filter interface. It is used to construct simple Filter. This abstract class provides some common functions shared among concrete simple filter classes.

AbstractAssociativeFilter (abstract class):
This abstract class is a concrete implementor of the Filter interface. It is used to construct AssociativeFilter. This abstract class also provides concrete addFilter() method to add a filter to the AssociativeFilter.

AndFilter (class):
This class extends AbstractAssociativeFilter. It is used to construct “AND” search criterion.

OrFilter (class):
This class extends AbstractAssociativeFilter. It is used to construct “OR” search criterion.

NotFilter (class):
This class is a concrete implementer of the Filter interface. It is used to construct “NOT” search criterion.

InFilter (class):
This class is a concrete implementer of the Filter interface. It is used to construct “IN” search criterion.

GreaterThanFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “GreaterThan” search criterion. This class also provides concrete isValid(), and getFilterType() methods.

GreaterThanorEqualToFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “GreaterThanOrEqualTo” search criterion. This class also provides concrete isValid(), and getFilterType() methods.

LessThanFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “LessThan” search criterion. This class also provides concrete isValid(), and getFilterType() methods.

LessThanOrEqualToFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “LessThanOrEqualTo” search criterion. This class also provides concrete isValid(), and getFilterType() methods.

EqualToFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “EqualTo” search criterion.

BetweenFilter (class):
This class extends AbstractSimpleFilter class. It is used to construct “Between” search criterion.

LikeFilter (class):
This class is a concrete implementer of the Filter interface. It is used to construct “Like” search criterion it supports both substring and wildcard searching.

NullFilter (class):
This class is a concrete implementer of the Filter interface. It is used to construct “Like” search criterion it supports both substring and wildcard searching.

Package com.topcoder.search.builder.database

DatabaseConnectionStrategy (class):
The class extends ConnectionStrategy class, and provides concrete implementations of abstract methods. This class is used to search database. It is also dedicated to one database connection. Once the connection is established, it can not be changed.
This class has been deprecated.

DatabaseConnectionInformation (class):
The class implements ConnectionInformation interface. This class includes all the information needed to connect to a database.
This class has been deprecated.

DatabaseSearchStringBuilder (class):
This class is a concrete implementer of SearchStringBuilder interface. It builds sql query string according to SQL92 grammar.
This class has been deprecated.

DatabaseSearchStrategy
This is a Search Strategy that is tuned for searching a database. It is responsible for building the necessary SQL search string appropriate to the filters provided and executing the SQL against the database. This is done with the help of the SearchFragmentBuilder implementations that are provided in this package. Each SearchFragmentBuilder is responsible for building the SQL for a specific filter, and a ClassAssociator is used to associate the FragmentBuilders with the filters (making the filter - Fragment mapping easier).

AndFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the AndFilter.

LikeFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the LikeFilter.

OrFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the OrFilter.

EqualsFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the EqualsFilter.

NotFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the NotFilter.

RangeFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the GreaterThanFilter, LessThanFilter, GreaterThanOrEqualToFilter, LessThanOrEqualToFilter, BetweenFilter.

NullFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the NullFilter.

InFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the InFilter.

Package com.topcoder.search.builder.ldap

LDAPConnectionStrategy (class):
The class extends ConnectionStrategy class, and provides concrete implementations of abstract methods. This class is used to search LDAP directory. It is also dedicated to one LDAP connection. Once the connection is established, it can not be changed.
This class has been deprecated.

LDAPSearchStringBuilder (class):
This class is a concrete implementer of SearchStringBuilder interface. It builds LDAP filter string according to LDAP filter syntax.
This class has been deprecated.

LDAPConnectionInformation (class):
The class implements ConnectionInformation interface. This class includes all the information needed to connect to a LDAP directory.
This class has been modified slightly to return an LDAPSDK in getFactory method instead of Object.

LDAPSearchStrategy
This is a Search Strategy that is tuned for searching an LDAP data store. It is responsible for building the necessary LDAP search string appropriate to the filters provided and executing the search string on the LDAP server. This is done with the help of the SearchFragmentBuilder implementations that are provided in this package. Each SearchFragmentBuilder is responsible for building the LDAP search string fragment for a specific filter, and a ClassAssociator is used to associate the FragmentBuilders with the filters (making the filter - Fragment mapping easier).

AndFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the AndFilter.

LikeFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the LikeFilter.

OrFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the OrFilter.

EqualsFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the EqualsFilter.

NotFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the NotFilter.

RangeFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the GreaterThanFilter, LessThanFilter, GreaterThanOrEqualToFilter, LessThanOrEqualToFilter, BetweenFilter.

NullFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the NullFilter.

InFragmentBuilder
This FragmentBuilder is used to handle building the SearchContext for the InFilter.

1.5 Component Exception Definitions

NullPointerException:
This represents some nullable field (String, object, etc) that was passed to a function that cannot handle null values.

Almost (there are some exceptions) any class that deals with a String or Object will throw this exception when a null value is encountered. The methods that throw this are clearly marked in the tags section of the documentation tab.

To comply with recent Topcoder standards, the component should now throw IllegalArgumentException for null arguments.

IllegalArgumentException:
This exception is thrown when the parameters passed to a function is an empty string, an empty Set, or an empty Map. It will be also thrown if the parameters do not match. Almost (there are some exceptions) any class that deals with a String, a Set or a Map will throw this exception. The methods that throw this are clearly marked in the tags section of the documentation tab.

OperationNotSupportedException:
This is a custom exception. It extends the SearchBuilderException. It is thrown if the intended operation is not supported. For example, if SearchBundle is bounded to LDAP, but user wants to call compileStatement() (which is used to compile statement for database). Then this exception is thrown.

SearchBuilderConfigurationException:
This is a custom exception. It extends the SearchBuilderException. It is thrown if any error occurs when reading the configuration file. ConfigurationManager exceptions should be wrapped by this exception and then thrown.

DuplicatedElementsException
This is a custom exception. It extends the SearchBuilderException. It is thrown by SearchBundleManager, if clients try to add a SearchBundle that already exists

PersistenceOperationException
This is a custom exception. It extends the SearchBuilderException. It is thrown if connection to data store fails, or any operations over data store fails. Any exceptions generated by either LDAP or database must be wrapped by this exception and then thrown.

UnrecognizedFilterException
This custom exception is thrown if a SearchStrategy or SearchFragmentBuilder is unable to recognize the provided filter.

SearchBuilderException:
This is a custom exception. All the other custom exceptions defined for this component

extends this exception. This exception extends com.topcoder.util.errorhandling.BaseException

1.6 Thread Safety

One of the requirements in this version explicitly asked for each search to be performed on a different connection in order to support multiple concurrent requests. The design respects this requirement, and the new SearchStrategies make use of a State object to build up each request. This ensures that the component is able to function smoothly even when multiple threads are using the Search methods.

However, it would be overkill to implement complete thread safety by immutability, and synchronization would be too much of an overhead. Read-write lock also seems like an unnecessary effort, given that the mutable attributes of the SearchBundle will likely not be changed once the SearchBundle has been deployed. As such, the client is simply warned not to call any mutating methods on the SearchBundle and only use it to initialize the bundle.

2. Environment Requirements

2.1 Environment

- Java 1.4

2.2 TopCoder Software Components

- Configuration Manager
- DB Connection Factory
- LDAP SDK Interface
- Database Abstraction
- Data Validation
- Base Exception
- Class Associations
- Object Factory

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.search.builder

3.2 Configuration Parameters

Parameters for SearchBundleManager

Parameter	Description	Values
searchStrategyFactor	The name of the SpecificationFactory	Valid String.

yNamespace	used to create the Object Factory used for creating the SearchStrategy implementations.	
fieldValidatorFactory Namespace	The name of the SpecificationFactory used to create the Object Factory used for creating the ObjectValidator implementations used to validate the searchable fields.	Valid String.
searchBundles	This property is a container for nested properties providing the details for configured bundles.	N/A
searchBundle.<boundleName>	This property is a container for nested properties providing the configuration parameters for a particular search bundle. The name of this property represents the name of configured bundle and should be unique across entire configuration. This is the name under which the bundle can be acquired from the BundleManager	N/A
searchBundle.<boundleName>.searchableFields	This class will contain the subproperties which contain the fields which may be searched, and any validators attached to it.	Valid string
searchBundle.<boundleName>.searchableFields.<fieldName>	This property provides the name of a searchable field. **Optional.**	N/A
searchBundle.<boundleName>.searchableFields.<fieldName>.validator.class	This property provides the name of the class from which to retrieve an Object Validator to use from the Object Factory. **Required if the validator subproperty exists. The Validator Subproperty is optional.**	com.topcoder.util.datavalidator.NotValidator
searchBundle.<boundleName>.searchableFields.<fieldName>.validator.identifier	This property provides an optional identifier from which to retrieve an Object Validator to use from the Object Factory. **Optional.**	notNullValidator
searchBundle.<boundleName>.searchStrategy	The subproperty contains additional properties to specify the construction of the search strategy. **Required.**	N/A
searchBundle.<boundleName>.searchStrategy.class	The subproperty contains the classname of the SearchStrategy to retrieve from the Object Factory. **Required.**	com.topcoder.search.builder.database.DBSearchStrategy
searchBundle.<boundleName>.searchStrategy.identifier	The subproperty contains the optional identifier of the SearchStrategy to retrieve from the Object Factory. **Optional.**	oracleSearchStrategy.
searchBundle.<boundleName>.alias	The subproperties of this property provide the alias mapping to use. **Optional**	N/A
searchBundle.<boundleName>.alias.<aliasName>	The name of the property establishes the name of the alias. The value of the property is the actual searchable field as it exists in the data store. **Optional.**	Valid string
searchBundle.<boundleName>.context	The value of this property is the search context for the bundle. **Required.**	Valid String.

Parameters for DatabaseSearchStrategy

Parameter	Description	Value
connectionFactory	The subproperties of this will contain the information on the DB connectionFactory to use. **Required.**	**N/A**
connectionFactory.name	The namespace to provide when instantiating the connection factory. **Required.**	**N/A**
connectionFactory.class	The class of the connectionFactory to instantiate. This is optional and defauls to DBConnectionFactoryImpl. **Optional.**	**N/A**
connectionName	The name to request from the connection factory when acquiring a connection. If not present, then the default connection is used. **Optional.**	**Valid String.**
searchFragmentFactory Namespace	The namespace of the SpecificationFactory to create the Object Factory which will create the SearchFragmentBuilders. **Required.**	**Valid String.**
searchFragmentBuilders	This property will contain subproperties from which to retrieve the searchFragmentBuilders to use. **Optional.**	**N/A**
searchFragmentBuilder. <fragmentName>	The name may be anything. Subproperties of this property will contain the information regarding a fragment builder to use.	N/A
searchFragmentBuilder. <fragmentName>. targetFilter	The name of the target filter for which this SearchFragmentBuilder corresponds to. **Required.**	**Valid String.**
searchFragmentBuilder. <fragmentName>. classname	The fully-qualified classname of the searchFragmentBuilder to use. This value will be provided to the ObjectFactory to create the	**Valid String.**

	SearchFragmentBuilder. **Required.**	
searchFragmentBuilder. <fragmentName>. identifier	An optional identifier to provide to the searchFragmentBuilder to use. This value will be provided to the ObjectFactory to create the SearchFragmentBuilder. **Optional.**	**Valid String**

Parameters for LDAPSearchStrategy

Parameter	Description	Value
searchFragmentFactory Namespace	The namespace of the SpecificationFactory to create the Object Factory which will create the SearchFragmentBuilders. **Required.**	**Valid String.**
searchFragmentBuilders	This property will contain subproperties from which to retrieve the searchFragmentBuilders to use. **Optional.**	**N/A**
searchFragmentBuilder. <fragmentName>	The name may be anything. Subproperties of this property will contain the information regarding a fragment builder to use.	N/A
searchFragmentBuilder. <fragmentName>. targetFilter	The name of the target filter for which this SearchFragmentBuilder corresponds to. **Required.**	**Valid String.**
searchFragmentBuilder. <fragmentName>. classname	The fully-qualified classname of the searchFragmentBuilder to use. This value will be provided to the ObjectFactory to create the SearchFragmentBuilder. **Required.**	**Valid String.**
searchFragmentBuilder. <fragmentName>. identifier	An optional identifier to provide to the searchFragmentBuilder to use. This value will be provided to the ObjectFactory to create the SearchFragmentBuilder. **Optional.**	**Valid String**
connectionInfoFactory Namespace	The namespace of the SpecificationFactory to create the Object Factory which will create the ConnectionInfo. **Required.**	**Valid String.**
connectionInfo	This will contain the subproperties needed to create the	**N/A**

	LDAPConnectionInformation object **Required.**	
connectionInfo.classname	The name of the class from which to retrieve the LDAPConnectionInformation from the Object Factory. It is optional and defaults to com.topcoder.search.builder.ldap.LDAP ConnectionInformation. **Optional.**	**Valid String.**
connectionInfo.identifier	The identifier to use when retrieving the LDAPConnectionInformation from the Object Factory. **Optional.**	**Valid String.**

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

Note to Developer: This portion may need to be updated to reflect any new test cases or configuration required to test the new 1.3 version of Search Builder.

- Extract the component distribution.
- Set up a database, and initialize the tables with test_files/mysql.sql and test_files/mysql12.sql (you may need to adapt the schema to your database).
- Configure the connection info within:
    - test_files/config.xml
    - test_files/Database.xml
    - test_files/failuretests.xml
    - test_files/stresstests/DBConfig.xml
- Set up an LDAP server. The rest of the configuration information will assume you use OpenLDAP, you may need to adapt the details to your LDAP server.
- Create the following LDAP schema (name it searchbuilder.schema):

```
attributetype ( 2.5.4.59
    NAME 'searchbuild'
    DESC 'RFC2256: for searchbuilder test'
    EQUALITY caseIgnoreMatch
    ORDERING caseIgnoreOrderingMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{128} )

objectclass ( 2.5.6.51
    NAME 'role'
    DESC 'RFC2256: a role'
```

```
SUP top STRUCTURAL
MUST (cn $ searchbuild)      )
```

- Edit slapd.conf to include the above schema, make the server listen on localhost port 389, and have the following configuration:

```
suffix          "dc=guessant,dc=org"
rootdn          "cn=Manager,dc=guessant,dc=org"
rootpw          secret
```

- Add an initial entry. You can do so in the following way:
  - Create a file called init.ldif with the following content:

```
dn: dc=guessant,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
o: guessant
dc: guessant
```

  - After starting the server, run the following command:

```
ldapadd -a -x -h localhost -D cn=Manager,dc=guessant,dc=org -w
secret -f init.ldif
```

- Start the LDAP server if you haven't already done so.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

The most like usage scenario to get a SearchBundle object could look like as follows:

For SearchBundleManager initialized from the parameters specified by the configuration file.

a. Create a configuration file providing all required parameters.
b. Add an entry for the above file to com/topocder/util/config/ConfigManager.properties file
c. Instantiate the SearchBundleManager providing the configuration namespace.
d. Retrieve a single SearchBundle object from SearchBundleManager.
e. Perform various operations over the SearchBundle object.

4.3 Demo

The code below demonstrates the usage scenario mentioned above.

Sample Configuration Files have been provided in the docs directory. The following files are provided:

+ *docs/dbSearchStrategyConfig.xml* contains the configuration with the Class Associations that will be used for the DBSearchStrategy of this release. The Class Associations that are configured here will serve as an example on how to map the SearchFragmentBuilders to the Filter implementations.

+ *docs/ldapSearchStrategy.xml* contains the configuration with the Class Associations that will be used for the LDAPSearchStrategy of this release. The Class Associations that are configured here will serve as an example on how to map the SearchFragmentBuilders to the Filter implementations.

4.3.1 First Usage Scenario

4.3.1.1 Demo usage of the search database via `searchBundle`

```
        //get the dbsearchBundle
        SearchBundle dbsearchBundle =
manager.getSearchBundle("FirstSearchBundle");

        //get the people whose age = 10
        EqualToFilter equalToFilter = new EqualToFilter("The age",
                new Integer(10));

        //search depend the filter
        CustomResultSet result = (CustomResultSet)
dbsearchBundle.search(equalToFilter);

        List list = new ArrayList();
        //get the result
        while (result.next()) {
            list.add(result.getString("peopleName"));
        }
        assertEquals("There are 2 prople with age = 10", 2, list.size());
        //makeup values
        List values = new ArrayList();
        values.add(new Integer(3));
        values.add(new Integer(5));

        //construct infilter
        InFilter infilter = new InFilter("The age", values);
        result = (CustomResultSet) dbsearchBundle.search(infilter);
        int siz = 0;
        while (result.next()) {
            siz++;
        }
        assertEquals("3 items are live up to the infilter.", 3, siz);

        NotFilter notFilter = new NotFilter(infilter);
        AndFilter andFilter = new AndFilter(infilter, notFilter);
        result = (CustomResultSet) dbsearchBundle.search(andFilter);
        siz = 0;

        while (result.next()) {
            siz++;
        }

        assertEquals("0 items are live up to the infilter.", 0, siz);

        list = new ArrayList();
```

4.3.1.3 Demo usage of the search ldapserver via searchBundle

```
       //get the dbsearchBundle
        SearchBundle ldapsearchBundle = manager.getSearchBundle(
                "SecondSearchBundle");
```

```
        ldapsearchBundle.setSearchableFields(fields);

        EqualToFilter equalToFilter1 = new EqualToFilter("sn", "5");
        EqualToFilter equalToFilter2 = new EqualToFilter("sn", "3");
        EqualToFilter equalToFilter3 = new EqualToFilter("sn", "0");
        //get the prople who sn = "5"
        Iterator it = (Iterator) ldapsearchBundle.search(equalToFilter1);

        AndFilter andFilter = new AndFilter(equalToFilter1, equalToFilter2);
        it = (Iterator) ldapsearchBundle.search(andFilter);

        OrFilter orFilter = new OrFilter(equalToFilter1, equalToFilter2);
        orFilter.addFilter(equalToFilter3);
        it = (Iterator) ldapsearchBundle.search(orFilter);

        //2 peoples with name = "0" | "3" | "5"
        assertEquals("There are 2 people under the condition setted", 2,
                TestHelper.getItemNumber(it));

        NotFilter notFilter = new NotFilter(equalToFilter1);
        it = (Iterator) ldapsearchBundle.search(notFilter);

        //8 items with name != "5"
        assertEquals("There are 8 people under the condition setted", 8,
                TestHelper.getItemNumber(it));
        List values = new ArrayList();
        values.add("3");
        values.add("4");
        BetweenFilter bwtweenFilter = new BetweenFilter("sn", "6", "7");
        it = (Iterator) ldapsearchBundle.search(bwtweenFilter);

        InFilter inFilter = new InFilter("sn", values);
        it = (Iterator) ldapsearchBundle.search(inFilter);

        GreaterThanFilter greaterThanFilter = new GreaterThanFilter("sb",
"sb1");
        it = (Iterator) ldapsearchBundle.search(greaterThanFilter);

        LessThanFilter lessThanFilter = new LessThanFilter("sb", "sb1");
        it = (Iterator) ldapsearchBundle.search(lessThanFilter);
```

4.3.2 Demo For version 1.2

4.3.2.1 The demo usage of create LikeFilter.

```
    / /create LikeFilter via the contructor with name and value
        likeFilter = new LikeFilter(NAME, LikeFilter.CONTAIN_TAGS + VALUE);

        //create LikeFilter via the contructor with name, value and escapeChar
        likeFilter = new LikeFilter(NAME, LikeFilter.CONTAIN_TAGS + VALUE, '!');

        //create the likeFilter via SearchBundle
        likeFilter = new LikeFilter(NAME,
                LikeFilter.CONTAIN_TAGS + VALUE);
        likeFilter = new LikeFilter(NAME,
                LikeFilter.CONTAIN_TAGS + VALUE, '!');
```

4.3.2.2 The demo usgae of search database with the LikeFilter

```
likeFilter = new LikeFilter("name", LikeFilter.CONTAIN_TAGS + "ac");

        //search all the items whose name contains 'ac' subString
        CustomResultSet result = (CustomResultSet)
dbSearchBundle.search(likeFilter);
```

```
        //search all the items whose name start with 'ac' subString
        likeFilter = new LikeFilter("name", LikeFilter.START_WITH_TAG + "ac");
        result = (CustomResultSet) dbSearchBundle.search(likeFilter);

        //search all the items whose name end with 'ac' subString
        likeFilter = new LikeFilter("name", LikeFilter.END_WITH_TAG + "ac");
        result = (CustomResultSet) dbSearchBundle.search(likeFilter);

        //search all the items
        likeFilter = new LikeFilter("name", LikeFilter.WITH_CONTENT + "%");
        result = (CustomResultSet) dbSearchBundle.search(likeFilter);

        //search the likeFilter with the otherFilters, such as andFilter,
orFilter and NotFilter
        Filter anotherFilter = new GreaterThanFilter("age", "15");
        Filter andFilter = new AndFilter(likeFilter, anotherFilter);
        result = (CustomResultSet) dbSearchBundle.search(andFilter);

        Filter orFilter = new OrFilter(likeFilter, anotherFilter);
        result = (CustomResultSet) dbSearchBundle.search(orFilter);
```

4.3.2.3 The demo usgae of search LDAP with the LikeFilter.

```
likeFilter = new LikeFilter("searchbuild",
               LikeFilter.CONTAIN_TAGS + "r5");

        //search all the entrys whose searchbuild attribute contains 'r5'
subString
        Iterator result = (Iterator) ldSearchBundle.search(likeFilter);

        //search all the entrys whose searchbuild attribute start with 'like'
subString
        likeFilter = new LikeFilter("searchbuild",
                LikeFilter.START_WITH_TAG + "like");
        result = (Iterator) ldSearchBundle.search(likeFilter);

        //search all the entrys whose searchbuild attribute end with 'ac'
subString
        likeFilter = new LikeFilter("searchbuild",
                LikeFilter.END_WITH_TAG + "ac");
        result = (Iterator) ldSearchBundle.search(likeFilter);

        //search all the entrys
        likeFilter = new LikeFilter("searchbuild", LikeFilter.WITH_CONTENT +
                "*");
        result = (Iterator) ldSearchBundle.search(likeFilter);

        //search the likeFilter with the otherFilters, such as andFilter,
orFilter and NotFilter
        Filter anotherFilter = new GreaterThanFilter("cn", "s5");
        Filter andFilter = new AndFilter(likeFilter, anotherFilter);
        result = (Iterator) ldSearchBundle.search(andFilter);

        Filter notFilter = new NotFilter(likeFilter);
        result = (Iterator) ldSearchBundle.search(notFilter);
```

4.3.3 Demo For version 1.3 (Null Filters)

```
       // Search for all people without a taxForm present.
       NullFilter nullFilter = new NullFilter(taxForm);

       CustomResultSet customResultSet = (CustomResultSet)
searchBundle.search(nullFilter);
```

5. Future Enhancements

Future enhancements include:
Add more sub-components to support more types of data store.
Support optional table joins.