JNDI Context Utility 2.0 Component Specification

Please note that changes to version 1.0 are shown in **blue** font and new additions are shown in **red** font.

1. Design

The `JNDI Context Utility` component is a simple component providing static utility
methods to manipulate JNDI Contexts. The functionality provided by the component is the following:

- Recursively create subcontexts within a specified Context.
- Obtain a JMS Queues and Topics from specified Context
- Obtain an SQL Connection from a JNDI DataSource
- Get the remote objects registered in remote object registry
- Perform conversion between Strings and JNDI Names
- Create a JNDI Name from a String that is valid within a specified Context
- Dump the Content of specified Context or its subcontext as an XML Document to standard output or represent a Context as an XML Document
- Manually or programmatically save the Contexts initialization parameters in configuration file
- Create a Context using the name corresponding to initialization parameters stored in configuration file

The `JNDI Context Utility` component provides a flexible way to get a representation of a
Context. It provides an interface named `ContextRenderer`. This interface represents a listener that receives notifications of events such as CONTEXT FOUND, BINDING FOUND and CONTEXT ENDED.
The component includes two implementations of `ContextRenderer`: `ContextConsoleRenderer` and `ContextXMLRenderer`. The first one allows provides a utility to dump the Context to standard output using a command line interface and the second one allows access to an XML Document object representing the Context. Both classes represent a Context using following XML DTD :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Context (Context*, Binding*)>
<!ATTLIST Context
name CDATA #IMPLIED
fullname CDATA #IMPLIED
>
<!ELEMENT Binding (Name, Class)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Class (#PCDATA)>
```

1.0.1 Version 2.0 changes

In version 2.0 of the component we will add the ability to create instances of JNDIUtil classes. This will allow for each instance to have its own specific Context instance which allows for much more flexible configuration. Previous version was based on a default context for most of its API. The API is virtually the same as in the JNDIUtils utility class (as provided by static methods) so there is little development effort needed here.
In addition to this, new way of configuring the component has been introduced. In addition to the 1.0 version which used Configuration Manager we will also use a simple xml file with the following simple format:

```
<Properties>
     <Property name=propertyName>value</Property>
     <Property name=propertyName>value</Property>
    . . .
     <Property name=propertyName>value</Property>
<Properties>
```

This is basically it. This way we can configure any properties that we would like without dependencies on Config Manager. Here is the XSD for the simple configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--W3C Schema generated by XMLSpy v2007 (http://www.altova.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:simpleType name="ST_Property"></xs:simpleType>
	<xs:complexType name="CT_Property">
		<xs:simpleContent>
			<xs:extension base="ST_Property">
				<xs:attribute name="name" use="required">
					<xs:simpleType>
						<xs:restriction base="xs:string">
						</xs:restriction>
					</xs:simpleType>
				</xs:attribute>
			</xs:extension>
		</xs:simpleContent>
	</xs:complexType>
	<xs:complexType name="CT_Properties">
		<xs:sequence>
			<xs:element ref="Property" maxOccurs="unbounded"/>
		</xs:sequence>
	</xs:complexType>
	<xs:element name="Property" type="CT_Property"/>
	<xs:element name="Properties" type="CT_Properties"/>
	</xs:schema>
```

1.1 Design Pattern

`JNDIUtils` class implements a ***`Utility` pattern***. The combination of `JNDIUtils` and `ContextRenderer` is similar to a ***`Listener` pattern*** (the `ContextRenderer` listens on events from `JNDIUtils.dump()` methods that notifies the `ContextRenderer` about content of the Context).

JNDIUtil class uses ConfigurationStrategy in a bit like a ***Strategy pattern***. Note that while the strategy is not directly pluggable it is 'swapped' based on the type of constructor input.

1.2 Industry Standards

None.

1.3 Required Algorithms

1) Dumping the Context and subcontexts

```
NamingEnumeration enum = ctx.list("");
// notify renderer on start of context
renderer.startContext(ctx.getNameInNamespace(), relativeName);
while (enum.hasMore()) {
pair = (NameClassPair) enum.next();
if (mode) {
try {
next = (Context) ctx.lookup(pair.getName());
dump(next, renderer, pair.getName(), mode);
} catch(ClassCastException e) {
renderer.binding(pair.getName(), pair.getClassName());
}
} else {
// notify renderer on binding
renderer.bindingFound(pair.getName(), pair.getClassName());
}
```

```
}
// notify renderer on end of context
renderer.endContext(ctx.getNameInNamespace(), relativeName);
```

2) Creating Context with initial parameters from configuration file

```
Properties props = new Properties();
String url = null;
String factory = null;
name = "context." + name + ".";
factory = cm.getString(NAMESPACE, name + "factory");
if (factory == null) {
throw new NamingException("Factory not specified");
}
props.put(Context.INITIAL_CONTEXT_FACTORY, factory);
url = cm.getString(NAMESPACE, name + "url");
if (url != null) {
props.put(Context.PROVIDER_URL, url);
}
return new InitialContext(props);
```

3) Saving Context initial parameters to configuration file

```
cm.createTemporaryProperties(NAMESPACE);
name = "context." + name + ".";
Enumeration enum = props.propertyNames();
while (enum.hasMoreElements()) {
String pname = (String) enum.nextElement();
cm.setProperty(NAMESPACE,
name + pname, (String) props.get(pname));
}
cm.commit(NAMESPACE, "JNDIUtils");
```

4) Get the object from Context verifying its type

```
Object object = ctx.lookup(name);
return (The target class to cast) object;
```

5) Create Name object from String

```
Properties props = new Properties();
props.put("jndi.syntax.separator", "/");
props.put("jndi.syntax.direction", "left_to_right");
return new CompoundName(name.replace(separator, '/'), props);
```

6) Create Name form String that is valid within Context

```
NameParser parser = ctx.getNameParser("");
return parser.parse(name);
```

1.4 Component Class Overview

JNDIUtils
This is the main class of the component and provides the majority of the functionality. The functions are all public static methods only. Most of the methods accept a Context to perform the operations on. The class contains overloaded methods that accept both Strings and JNDI Names as names of contexts.

ContextRenderer
An interface representing the renderer of the content of the Context. The classes implementing this interface acts as a listener on the content of the context. The methods of this interface are invoked from `JNDIUtils.dump()` methods to notify an implementation of `ContextRenderer` on the content of the `Context`. The methods of

this interface are invoked when a `Context`, subcontext or binding is found or Context has ended.

ContextConsoleRenderer
An implementation using the command line interface of ContextRenderer used to display an XML representation of the Context or subcontext of the specified Context to standard output.

ContextXMLRenderer
An implementation of ContextRenderer used to retrieve an XML Document object corresponding to specified Context.

JNDIUtil
This is a version of the utility JNDIUtils functionality but which is instance based. This means that each instance can have its specific Context used (based on either a setter or through configuration)
Other than that (i.e. instance of Context) the functionality is exactly the same as in the JNDIUtils class with the other main difference being that in addition to the Config Manager (supported by JNDIUtils), this class also supports alternative XML configuration (which mimics the Config Manager in its scope)

ConfigurationStrategy
This is a generic interface for simple configuration functionality that is used by this component. We basically have the ability to fetch a named property value and we have the ability to write/overwrite a specific property and then commit it to the configuration medium.

ConfigurationManagerConfigurationStrategy
This is a specific implementation of the ConfigurationStrategy interface contract for dealing with ConfigurationManager.
Here we will be able to read/write and commit string properties.

XmlFileConfigurationStrategy
This is a specific implementation of the ConfigurationStrategy interface contract for dealing with am xml file with configuration data.
Here we will be able to read/write and commit string properties.
Note that we use a validating parser if the user requests that.

1.5 Component Exception Definitions

java.lang.IllegalArgumentException
Thrown from any methods of `JNDIUtils`, `ContextRenderer`, `ContextConsoleRenderer` and `ContextXMLRenderer` if any specified parameter is null.

javax.naming.NamingException
Thrown from almost all methods of `JNDIUtils`, if any exception related to naming system occurs.

java.sql.SQLException
Thrown from `JNDIUtils.getConnection()` methods if any SQL error occurs while getting SQL connection from specified `DataSource`.

java.lang.ClassCastException

Thrown from `JNDIUtils.getObject()` methods verifying that object found in `Context` can be cast to specified type if object can not be cast to specified type.

com.topcoder.util.config.ConfigManagerException
Is thrown from `JNDIUtils's getDefaultContext(), getContext()` and `saveContextConfig()` methods if any exception related to Configuration Manager occurs while retrieving or storing initialization parameters of Contexts in configuration file.

javax.naming.InvalidNameException
Is thrown from `JNDIUtils.createName(String,char)` if the Name object can not be created from given String.

ConfigurationException
This is a custom exception used to signal issues with configuration data.

1.6 Thread Safety

First, please note there are a couple methods that update the configuration source (Config Manager or the input xml file) In this sense there is no thread safety since we cannot really lock these files in a deterministic manner.
In a practical manner the application API is thread safe in the sense that both the JNDIUtils utility and the JNDIUtil class have no mutable state.

2. Environment Requirements

2.1 Environment

None.

2.2 TopCoder Software Components:

Configuration Manager 2.1.5: Used to specific configuration data for the component.
Base Exception 2.0: Used to create a base exception for all custom exception. The 1.0 has been refactored slightly to accommodate this without any changes to external API.

2.3 Third Party Components:

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.naming.jndiutility
com.topcoder.naming.jndiutility.configstrategy

3.2 Configuration Parameters

com/topcoder/naming/jndiutility/JNDIUtils.properties
This file is placed into a directory accessible from the CLASSPATH. It should contain the initializing parameters of Contexts that may be created by JNDIUtils by names. The format of records in such file is as follows :

```
context.<context_name>.<property> = <value>, for example :
context.default.factory=com.sun.jndi.ldap.LdapCtxFactory
context.default.url=ldap://openldap.org:389/dn=OpenLDAP,dn=org
```

If such Context is specified it then can be retrieved with JNDIUtils.getContext("default") method.
The `factory` and `url` property names are reserved for use by `JNDIUtils` class. Any additional properties may be specified.
For each context specified in configuration file only `factory` property is required.

Note that for the solution that uses the xml format mentioned in section 1.0.1 the idea of the properties is the same. We still use the property name as above (for example context.default.factory) but we will read if from the xml configuration file directly instead of the mechanism provided by Configuration Manager.

3.3 Dependencies Configuration

In order to create contexts using initial parameters from configuration file the .jar files with classes corresponding to specified context factories should be placed in CLASSPATH. JNDI classes are part of Java 1.4 SDK .

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Create empty temporary directory that will be used during tests
- Modify `context.default.url` property
- Create another temporary directory to point to this directory
- Replicate following directory structure under it (suppose that this directory is c:\tmp):

```
c:\tmp
    |
    +--------dir
    |       |
    |       +---------- dirfile.txt
    +------------file.txt
```

- Modify `context.test.url` property to point to his directory
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

No special steps are needed.

4.3 Demo

```
/*
* Get the default context which initial parameters are stored in
* configuration file under context.default. name
*/
Context ctx = JNDIUtils.getDefaultContext();
/*
* Create given subcontext within given Context
*/
JNDIUtils.createSubcontext(ctx, "com/topcoder/naming/jndiutility");

/*
* Store initial parameters of Context in configuration file for
* further use.
*/
Properties props = new Properties();
props.put("factory", "com.sun.jndi.fscontext.RefFSContextFactory");
props.put("url", "file://c:\\com\\topcoder");
JNDIUtils.saveContextConfig("new_context", props);

/*
* Create context using initial parameters from configuration file.
*/
ctx = JNDIUtils.getContext("new_context");

/*
* Get the resources from Context
*
*/
```

```
Queue queue = JNDIUtils.getQueue(ctx, “MyQueue”);
Topic topic = JNDIUtils.getTopic(ctx, “MyTopic”);
Connection con = JNDIUtils.getConnection(ctx, “MyConnection”);

/*
* Get object from context
*/
SomeObject object = JNDIUtils.getObject(“com/topcoder/info”);

/*
* Get object verifying that it can be cast to specified class
*/
object = JNDIUtils.getObject(“com/topcoder/info”, object.getClass());

/*
* Get the XML Document object corresponding to Context without traversing
* through nested subcontexts
*/
ContextXMLRenderer renderer = new ContextXMLRenderer();
JNDIUtils.dump(ctx, renderer, true);
Document doc = render.getDocument();

/*
* Get the XML Document object corresponding to some subcontext of Context
* traversing through nested subcontexts
*/
ContextXMLRenderer renderer = new ContextXMLRenderer();
JNDIUtils.dump(ctx, “naming/jndiutility”, renderer, true);
Document doc = renderer.getDocument();

/*
* Convert a String to Name object that is a compound name separated with
* slashes
*/
Name name = JNDIUtils.createName(“com.topcoder.util.config”, ‘.’);
System.out.println(name); // Will print : com/topcoder/util/Config

/*
* Convert a Name to String separated with any desired character
*/
String string = JNDIUtils.createString(name, ‘?’);
System.out.println(string); // Will print : com?topcoder?util?Config

In order to dump a Context to standard output using command line interface
following command
should be typed :
java com.topcoder.naming.jndiutility.JNDIUtils [-d] contextname [subcontext],

where :

-d – is an optional switch specifying that nested subcontexts should be traversed
too. If is not specified
then directly nested subcontexts will be represented like a simple bindings and
indirectly nested
subcontexts will not be dumped at all.
contextname – a name of context specified in configuration file
subcontext – an optional name of subcontext to dump

Using the instance class is very much the same as the static utility with the main
difference being that we can deal with specific namespaces or with specific XML
file locations as well as being able to set a Context directly:


/*
* Create an instance of the JNDIUtil instance with configuration based on
*  1. default Config namespace
*  2. specific Config namespace
*  3. specific file (for XML) with validation
*  4. specific InputStream without validation
*/
JNDIUtil jndiUtilInstance = new JNDIUtil();
```

```
jndiUtilInstance = new JNDIUtil(“my.jndi.namespace”);
jndiUtilInstance = new JNDIUtil(new File(“c:\\myXMLFile”), true);
jndiUtilInstance = new JNDIUtil(new FileInputStream(“c:\\myXMLFile”), false);

/*
* Once we have an instance of the jndi util we can call instance methods (which
* are an exact subset of the JNDIUtils class so we will not show them again, but
* we will show some of them.
*/

// Get a queue based on a specific (configured context) and a JNDI name
jndiUtilInstance.getQueue(new CompositeName(some name));

// Get a connection based in specific instance context
jndiUtilInstance.getConnection(“my.connection.name”);
```

5.0 Future Enhancements

None at this point.