LDAP Integration Howto

Leo Przybylski
leo@rsmart.com

January 24, 2011

Contents

Abstract

In order to integrate a directory server over LDAP with KIM using Spring, there are preparations and steps to be handled. This document explains

- Steps for Implementation
- Steps for Integration into Rice
- Spring Configuration
- Properties setup
- DTO Reimplementations
- Spring Service Overrides

1 Steps for Implementing with KFS

These are the following steps to installing and configuring LDAP Integration for KFS.

1. Add rice-kim-ldap.jar to CLASSPATH

The easiest way to do this is to add the `rice-kim-ldap.jar` to `work/web-root/WEB-INF/lib/`. This will add the necessary Spring configuration and class files to your classpath.

2. Configure Spring with Directory Server Credentials

Create/Modify a spring-kim.xml

1. Create a `spring-kim.xml` file in your classpath
2. Make it look like this

Listing 1: spring-kim.xml

```
<bean id="contextSource"
    class="org.springframework.ldap.core.
        support.LdapContextSource">
    <property name="url" value="${rice.
        ldap.url}" />
    <property name="base" value="${rice.
        ldap.base}" />
    <property name="authenticationSource"
        ref="authenticationSource" />
</bean>
```

```
<bean id="authenticationSource"
  class="org.springframework.ldap.
      authentication.
      DefaultValuesAuthenticationSourceDecorator
      ">
  <property name="target" ref="
      springSecurityAuthenticationSource"
      />
  <property name="defaultUser" value="${
      rice.ldap.username}" />
  <property name="defaultPassword" value="
      ${rice.ldap.password}" />
</bean>
```

3. Configure your `spring-kim.xml` so that it points to your institution's directory server and base DN.
4. Add `spring-kim.xml` to institutional spring files in your `kfs-build.properties`. Below is an example.

Listing 2: kfs-build.properties

```
institution.spring.source.files=,com/
    rsmart/kim/spring-kim.xml
```

Configure Credentials

1. Add the following to the `build/external/security.properties`

Listing 3: build/external/security.properties

```
rice.ldap.username=${rice.ldap.username}
rice.ldap.password=${rice.ldap.password}
rice.ldap.url=${rice.ldap.url}
rice.ldap.base=${rice.ldap.base}
```

2. Add lines to your kfs-build.properties

Listing 4: kfs-build.properties

```
rice.ldap.username=your ldap user
rice.ldap.password=your ldap password
rice.ldap.url=your ldap url
rice.ldap.base=your ldap base dn
```

2 Steps for Integration into Rice

1. Checkout rice source code

The URL to checkout from is `https://test.kuali.org/svn/rice/branches/rice-release-1-`

2. Checkout Ldap Customization

The URL to checkout from is `https://svn.rsmart.com/svn/kuali/contribution/community/` into your rice path as `ldap`. The resulting structure would be `rice-release-1-0-3-br/ldap`

3. Add ldap module to rice pom.xml

```
<modules>
        <module>api</module>
        <module>impl</module>
        <module>ldap</module>
        <module>web</module>
        <module>sampleapp</module>
        <module>ksb</module>
        <module>kcb</module>
        <module>kns</module>
        <module>kim</module>
        <module>kew</module>
        <module>ken</module>
</modules>
```

3. Add LDAP as a dependency to web

Edit the `rice-release-1-0-3-br/pom.xml` and add the following:

```
<dependencies>
        <dependency>
                <groupId>${project.groupId}</
                    groupId>
                <artifactId>rice-impl</artifactId>
                <version>${project.version}</
                    version>
        </dependency>
        <dependency>
                <groupId>${project.groupId}</
                    groupId>
```

```
                <artifactId>rice-sampleapp</
                    artifactId>
                <version>${project.version}</
                    version>
        </dependency>
        <dependency>
                <groupId>${project.groupId}</
                    groupId>
                <artifactId>rice-ldap</artifactId>
                <version>${project.version}</
                    version>
        </dependency>
...
</dependencies>
```

2. Configure Spring with Directory Server Credentials

Modify a rice-config.xml

```
<param name="rice.ldap.username">uid=ldap,ou=App
     Users,dc=ldap,dc=rsmart,dc=com</param>
<param name="rice.ldap.password">6
    h5aXHLGCysQf3N4S9zYnuOtTijDVFZk</param>
<param name="rice.ldap.url">ldaps://ldap.rsmart.
    com:636</param>
<param name="rice.ldap.base">ou=People,dc=ldap,
    dc=rsmart,dc=com</param>

<param name="rice.additionalSpringFiles">org/
    kuali/rice/kim/config/KIMLdapSpringBeans.xml
    </param>
```

The following line is what enables the LDAP integration. Comment it out to disable integration.

```
<param name="rice.additionalSpringFiles">org/
    kuali/rice/kim/config/KIMLdapSpringBeans.xml
    </param>
```

3 Disabling/Enabling LDAP Integration

In order to disable the integration with LDAP, the method differs between KFS and Rice. Below are descriptions on the different methods.

3.1 KFS

You can remove the jar from the classpath.

3.2 Rice

Remove the KIMLdapSpringBeans.xml at build time.

4 Technical Details

4.1 Spring LDAP

Spring LDAP is an adapter layer between Spring and LDAP datasources.

The following description is taken from the *Spring LDAP* website:

> Spring LDAP is a Java library for simplifying LDAP operations, based on the pattern of Spring's JdbcTemplate. The framework relieves the user of common chores, such as looking up and closing contexts, looping through results, encoding/decoding values and filters, and more.
>
> The LdapTemplate class encapsulates all the plumbing work involved in traditional LDAP programming, such as creating a DirContext, looping through NamingEnumerations, handling exceptions and cleaning up resources. This leaves the programmer to handle the important stuff - where to find data (DNs and Filters) and what do do with it (map to and from domain objects, bind, modify, unbind, etc.), in the same way that JdbcTemplate relieves the programmer of all but the actual SQL and how the data maps to the domain model.
>
> In addition to this, Spring LDAP provides transaction support, a pooling library, exception translation from NamingExceptions to a mirrored unchecked Exception hierarchy, as

well as several utilities for working with filters, LDAP paths and Attributes.

Spring LDAP requires J2SE 1.4 or higher to run, and works with Spring Framework 2.0.x as well as 2.5.x. J2SE 1.4 or higher is required for building the release binaries from sources. Release 1.2.1 also requires an installation of JavaCC 4.0 when building from source. That is not necessary for release 1.3.x, since it uses Maven2, which handles all such dependencies behind the scenes.

To use it:

Listing 5: spring-datasource.xml

```
<beans>
    ...
    ...
    <bean id="contextSource"
        class="org.springframework.ldap.support.
            LdapContextSource">
        <property name="url" value="ldaps://ldap.
            rsmart.com:636" />
        <property name="base" value="ou=People,dc=
            com,dc=rsmart,dc=com" />
        <property name="userName" value="uid=<
            userid>,ou=App Users,dc=com,dc=rsmart,
            dc=com" />
        <property name="password" value="secret"
            />
        <property name="pool" value="true"/>
    </bean>
    <bean id="ldapTemplate" class="org.
       springframework.ldap.LdapTemplate">
        <constructor-arg ref="contextSource" />
    </bean>
    <bean id="ldapPrincipalDao"
        class="com.rsmart.kim.dao.LdapPrincipalDao
            ">
        <property name="ldapTemplate" ref="
            ldapTemplate" />
    </bean>
</beans>
```

```
\emph{Note that ldaps:// protocol is used.}
```

4.2 KIM

KIM interfaces need to be implemented within *KFS* that communicate over LDAP. *KIM* will delegate over LDAPs with *Spring LDAP* by implementing the following service interfaces.

4.2.1 IdentityService

Below is an description of which methods need to be overwritten to supply *KIM* with access to Person data from

getPrincipal
```
/** Get a KimPrincipal object based on the principalName. */
KimPrincipalInfo getPrincipal(String principalId);
```

getPrincipalByPrincipalName
```
KimPrincipalInfo getPrincipalByPrincipalName(String principalName);
```

lookupEntitys
```
/** Find entity objects based on the given criteria. */
List<KimEntity> lookupEntitys(Map<String, String> searchCriteria);
```

getEntityDefaultInfo
```
KimEntityDefaultInfo getEntityDefaultInfo( String entityId );
```

getEntityDefaultInfoByPrincipalId
```
KimEntityDefaultInfo getEntityDefaultInfoByPrincipalId( String principalId );
```

getEntityDefaultInfoByPrincipalName
```
KimEntityDefaultInfo getEntityDefaultInfoByPrincipalName( String principalName );
```

lookupEntityDefaultInfo

```
List<? extends KimEntityDefaultInfo>
lookupEntityDefaultInfo( Map<String,String>
 searchCriteria, boolean unbounded );
```

getMatchingEntityCount

```
int getMatchingEntityCount( Map<String,
String> searchCriteria );
```

getEntityPrivacyPreferences

```
KimEntityPrivacyPreferencesInfo
getEntityPrivacyPreferences( String
entityId );
```

getDefaultNamesForPrincipalIds

```
Map<String, KimEntityNamePrincipalNameInfo>
 getDefaultNamesForPrincipalIds(List<String
> principalIds);
```

getDefaultNamesForEntityIds

```
Map<String, KimEntityNameInfo>
getDefaultNamesForEntityIds(List<String>
entityIds);
```

4.2.2 UiDocumentServiceImpl

The `IdentityManagementPersonDocument` is still used to save modify role, group, and delegation assignments even though all entity information is coming through LDAP. This splits principal and entity information, but the `UiDocumentServiceImpl` makes it possible to accomplish this. The "Modify Entity" permission was removed from all roles because we no longer want entities to be managed through KFS.

Originally, the `UiDocumentServiceImpl` uses `Impl` domain objects couple to a database implementation, so it needs to be modified not to use uncoupled `Info` objects. Below is how `UiDocumentServiceImpl` is modified to do that.

loadEntityToPersonDoc is used to populate the `IdentityManagementPersonDocument` when the page loads from "edit" or "create new". Even though

entity information is not being stored in the database, it still needs to be present on persons.

saveEntityPerson is used to store the information and actually update the person. It needed to modified to take into consider the check for the "Modify Entity" permission. Normally, even if the permission isn't present, the document will try to save entity information. By checking for this permission, the desired behavior takes place which is entities will not be saved. Unlike `loadEntityToPersonDoc`, `Impl` domain objects are desirable here. The domain object that is modified is the `KimPrincipalImpl` which updates the `KRIM_PRNCPL_T` table and the necessary role, group, and delegation tables.

5 Development Steps

5.1 Setup Spring LDAP

5.1.1 Modifications to spring-kim.xml File

The following was added to connect *Spring LDAP*

```
<bean      id="contextSource"
        class="org.springframework.ldap.core.support
           .LdapContextSource">
    <property name="url" value="ldaps://ldap.
       rsmart.com:636" />
    <property name="base" value="ou=People,dc=ldap
       ,dc=rsmart,dc=com" />
    <property name="authenticationSource" ref="
       authenticationSource" />
</bean>

<bean      id="authenticationSource"
        class="org.springframework.ldap.
           authentication.
           DefaultValuesAuthenticationSourceDecorator
           ">
    <property name="target" ref="
       springSecurityAuthenticationSource" />
```

```
    <property name="defaultUser" value="uid=user,
        ou=App Users,dc=ldap,dc=rsmart,dc=com" />
    <property name="defaultPassword" value="[
        secret]" />
</bean>

<bean     id="springSecurityAuthenticationSource"
       class="org.springframework.security.ldap.
          SpringSecurityAuthenticationSource" />

<bean id="ldapTemplate" class="org.springframework
    .ldap.core.LdapTemplate">
     <constructor-arg ref="contextSource" />
</bean>
```

The *Kuali Rice* `ParameterService` is used to store the map between *KIM* and *LDAP* attributes. Still, many attribute names are stored in a constants class populated through Spring. See below

```
<bean id="kimConstants" class="org.kuali.rice.kim.
    util.ConstantsImpl">
  <!--
  <property name="kimLdapIdProperty"           value
      ="uaid" />
  <property name="kimLdapNameProperty"         value
      ="uid" />
  -->
  <property name="snLdapProperty"              value
      ="sn" />
  <property name="givenNameLdapProperty"       value
      ="givenName" />
  <property name="entityIdKimProperty"         value
      ="entityId" />
  <property name="employeeMailLdapProperty"    value
      ="mail" />
  <property name="employeePhoneLdapProperty"   value
      ="employeePhone" />
  <property name="defaultCountryCode"          value
      ="1" />
  <property name="mappedParameterName"         value
      ="KIM_TO_LDAP_FIELD_MAPPINGS" />
```

```
  <property name="mappedValuesName"              value
     ="KIM_TO_LDAP_VALUE_MAPPINGS" />
  <property name="unmappedParameterName"         value
     ="KIM_TO_LDAP_UNMAPPED_FIELDS" />
  <property name="parameterNamespaceCode"        value
     ="KR-SYS" />
  <property name="parameterDetailTypeCode"       value
     ="Config" />
  <property name="personEntityTypeCode"          value
     ="PERSON" />
  <property name="employeeIdProperty"            value
     ="emplId" />
  <property name="departmentLdapProperty"        value
     ="employeePrimaryDept" />
  <property name="employeeTypeProperty"          value
     ="employeeType" />
  <property name="employeeStatusProperty"        value
     ="employeeStatus" />
  <property name="defaultCampusCode"             value
     ="MC" />
  <property name="defaultChartCode"              value
     ="UA" />
  <property name="taxExternalIdTypeCode"         value
     ="TAX" />
  <property name="externalIdProperty"            value
     ="externalIdentifiers.externalId" />
  <property name="externalIdTypeProperty"        value
     ="externalIdentifiers.
     externalIdentifierTypeCode" />
  <property name="affiliationMappings"           value
     ="staff=STAFF, faculty=FCLTY, employee=STAFF,
     student=STDNT, affilate=AFLT"/>
  <property name="employeeAffiliationCodes"      value
     ="STAFF,FCLTY" />
</bean>
```

The constants class as well as the *Spring LDAP* integration and *Kuali Rice* `ParameterService` are injected into the `LdapPrincipalDaoImpl` instance.

```
<bean id="ldapPrincipalDao" class="org.kuali.rice.
    kim.dao.impl.LdapPrincipalDaoImpl">
  <property name="ldapTemplate"       ref="
      ldapTemplate" />
  <property name="parameterService" ref="
      parameterService" />
  <property name="kimConstants"       ref="
      kimConstants" />

</bean>
```

The `LdapPrincipalDaoImpl` is an implementation of `PrincipalDao` which is delegated by the `LdapIdentityServiceImpl`. The `LdapPrincipalDaoImpl` connects to *LDAP* and maps the principal and entity information into *KIM* domain objects.

2. Implement/Override Methods in IdentityService

3. Create PrincipalDao for searching for Principal/Entity information from LDAP.

5.1.2 Retrieving LDAP Information as KIM Domain Objects

Spring LDAP offers a `ContextMapper` interface for these kinds of mappings; therefore, all of the mappings are in pure java. This is how `KimPrincipal` is mapped from *LDAP*.

```
contextMappers.put(KimPrincipalInfo.class, new
    AbstractContextMapper() {
     public Object doMapFromContext(
         DirContextOperations context) {
          final KimPrincipalInfo person = new
              KimPrincipalInfo();
          person.setPrincipalId(context.
              getStringAttribute(getKimConstants().
              getUaidLdapProperty()));
          person.setEntityId(context.
              getStringAttribute(getKimConstants().
              getUaidLdapProperty()));
```

```
        person.setPrincipalName(context.
            getStringAttribute(getKimConstants().
            getUidLdapProperty()));
        return person;
    }
});
```

`contextMappers` is an instance map created for holding `ContextMapper` instances. Each DTO type has a mapper associated with it for retrieving the desired information from *LDAP*. Notice the use of `getKimConstants()`. This is how constant property names are used in the mapping. Also, notice that here the `ParameterService` is not used. The `ParameterService` is only used for mapping *KIM* criteria in lookup scenarios. When retrieving information from *LDAP*, the `ParameterService` is entirely useless. The `ContextMapper` is used instead. It gives more flexibility when mapping attributes of a specific class. Below is how the `ContextMapper` is actually used.

```
public <T> List<T> search(Class<T> type, Map<
    String, Object> criteria) {
    AndFilter filter = new AndFilter();

    for (Map.Entry<String, Object> entry :
        criteria.entrySet()) {
        if (entry.getValue() instanceof Iterable)
            {
            OrFilter orFilter = new OrFilter();
            for (String value : (Iterable<String>)
                entry.getValue()) {
                orFilter.or(new EqualsFilter(entry
                    .getKey(), value));
            }
            filter.and(orFilter);
        }
        else {
            filter.and(new EqualsFilter(entry.
                getKey(), (String) entry.getValue()
                ));
        }
    }
```

```
    return getLdapTemplate().search(
        DistinguishedName.EMPTY_PATH, filter.encode
        (), contextMappers.get(type));
}
```

Spring LDAP gives a very flexible API for querying Directory-Based systems. The `search()` method takes advantage of several classes from the API in order to create a fairly generic query of *LDAP*. On the last line, the `LdapTemplate` is used with a verb—ContextMapper—retrieved from the `contextMappers` map. It is retrieved by passing through the desired type; therefore, in the case of searching for a `KimPrincipal` we would use something like this:

```
public KimPrincipalInfo getPrincipal(String
    principalId) {
    Map<String, Object> criteria = new HashMap();
    criteria.put(getKimConstants().
        getKimLdapProperty(), principalId);
    List<KimPrincipalInfo> results = search(
        KimPrincipalInfo.class, criteria);

    if (results.size() > 0) {
        return results.get(0);
    }

    return null;
}
```

Again, there isn't any need for the `ParameterService` yet because we know exactly what we want from *LDAP*.

5.1.3 Using Mapping KIM Attributes to LDAP Attributes for Lookups

KIM has an API method called `lookupEntityDefaultInfo` which is used by Kuali Lookups for querying information. The call will provide a map of information in terms of *KIM* attributes. This means that the map or search criteria is pretty meaningless to *LDAP* or any Directory-based service for that matter. The *KIM* attributes need to be mapped to *LDAP* attributes in order for the query to be made. For this, the `ParameterService` is used.

```
public List<? extends KimEntityDefaultInfo>
    lookupEntityDefaultInfo(Map<String,String>
    searchCriteria, boolean unbounded) {
     List<KimEntityDefaultInfo> results = new
         ArrayList();
     Map<String, Object> criteria = new HashMap();

     for (Map.Entry<String, String> criteriaEntry :
          searchCriteria.entrySet()) {
          info(String.format("Searching with
              criteria %s = %s", criteriaEntry.getKey
              (), criteriaEntry.getValue()));

          if (isMapped(criteriaEntry.getKey())) {
               criteria.put(getLdapAttribute(
                   criteriaEntry.getKey()),
                   criteriaEntry.getValue());
          }
     }

     return search(KimEntityDefaultInfo.class,
         criteria);
}

private Matcher getKimAttributeMatcher(String
    kimAttribute) {
     Parameter mappedParam = getParameterService()
     .retrieveParameter(getKimConstants().
         getParameterNamespaceCode(),
     getKimConstants().getParameterDetailTypeCode()
         ,
     getKimConstants().getMappedParameterName());

     String regexStr = kimAttribute + "=([^=;]*)
         .*";
     return Pattern.compile(regexStr).matcher(
         mappedParam.getParameterValue());
}

private boolean isMapped(String kimAttribute) {
```

```
        return getKimAttributeMatcher(kimAttribute).
            matches();
    }

    private String getLdapAttribute(String
        kimAttribute) {
        Matcher matcher = getKimAttributeMatcher(
            kimAttribute);
        matcher.matches();
        return matcher.group(1);
    }
```

By using regular expressions and storing parameters in the database for retrieval by the `ParameterService`, the task of mapping *KIM* attributes to *LDAP* attributes is pretty trivial.