Developers and Deployers Guide to the Composite Group Service

Contents

Introduction

Early in the uPortal project, it became clear that some mechanism was needed for *grouping* portal users, chiefly for the purpose of authorization. The org.jasig.portal.groups package evolved in response. It defines an api for managing groups of portal *entities* such as IPersons and IChannels. The groups framework does not actually operate on IPersons or IChannels; it manipulates stub objects whose keys and types point to the underlying entities. The stubs are implementations of org.jasig.portal.groups.IEntity, and their only concern is their group memberships. A stub knows nothing about its underlying entity except its key and type. The groups it belongs to are implementations of org.jasig.portal.groups.IEntityGroup. The groups are recursive (groups can contain other groups) and homogeneous (their IEntities have only one type of underlying entity.)
Prior to version 2.1, groups came from a group service with a single store. With the release of version 2.1, uPortal ships with a composite group service made up of multiple component services, each with its own group store. This document describes the composite design and explains how to configure a composite group service for your environment. It is principally aimed at implementors and developers, but it may also be helpful to planners evaluating uPortal's support for native sources of group information. Although it is hidden from clients, the design of the composite group service is significant to the uPortal *Implementor* who must configure the composite service and to the uPortal *Developer* who may have to write a custom group service or group store to adapt a native source of group information to the portal groups design. The implementor assigns the appropriate role to each source of group information and represents the composition in an xml configuration file, while the group service or store developer must implement the various group api's.

What's In This Document?

The rest of the document is organized as follows: the first section (Goals and Rationale of the Composite Service) sets out the argument for a composite service. The next

section (The Service Design) describes the service api and and the service class hierarchy. This is followed by a discussion of group keys and their relationship to the composite design (Groups, Keys and Service Names) . These first 3 sections are aimed at developers, and if you are interested only in deploying a composite service, you can skim them or skip them entirely. The next 3 sections are aimed at deployers. They describe the process of composite service assembly (Assembling the Composite), the configuration file that controls the assembly process (Configuring the Composite) and the design of the LDAP group store that ships with uPortal 2.1 (The LDAP Group Store). The last section (Next Steps) presents a very general outline for getting started with groups.

Unless otherwise noted, all referenced classes are in the org.jasig.portal.groups package. I'm assuming some familiarity with the basic groups types, IGroupMember, IEntity and IEntityGroup, and with the service façade, org.jasig.portal.services.GroupService, which have changed little since uPortal 2.0. The best source for information about these types is javadoc for org.jasig.portal.groups. The following terms are used interchangeably: IEntityGroup and group; IGroupMember and group member; composite group service and service. Depending on the context, entity may refer to an IEntity or to the underlying entity that is referred to by the IEntity.

Goals and Rationale of the Composite Service

Many institutions have group information that is not under the control of the portal, more often than not in an LDAP server. In addition, some organizations have this information spread over a number of external sources. In order to use it, the portal needs a way to adapt external group information to the portal groups design and to combine group information from multiple sources. An LDAP adaptor, for instance, would let the portal recognize attributes in LDAP as group memberships. These memberships, supplemented by memberships stored in the reference portal database, could be associated with permissions that govern what a user could and could not do. In this way, portal authorization could be driven from LDAP and modified from within the portal.

The composite groups system is a framework for creating and managing these adaptors. Its job is to aggregate group information from a variety of sources and present it in a consistent format to clients who can remain unaware of its origins. In fact, group service clients never interact directly even with the composite group service. A client makes a request to the service façade to obtain a group member, and the group member acts as an entry point into the composite group system. Once the client has a reference to a new or pre-existing group member, it makes subsequent requests to the group member itself, and henceforth, the client can ignore the service of origin of any group member it navigates to.

The Service Design

The Service Hierarchy. The composite group service api is divided among 3 service types, IComponentGroupService, ICompositeGroupService and IIndividualGroupService, each of which defines responsibilities for a specific service role. An IComponentGroupService is a composite *component*. It is concerned with composite service assembly and with identifying components in the composite. The ICompositeGroupService represents the

composition as a whole, encapsulating service components and delegating requests for group services. The IIndividualGroupService defines responsibilities for a specific group service that reads and possibly writes groups. It is the *leaf component* in the composite. Together, they form the following class hierarchy:

IComponentGroupService

ICompositeGroupService extends IComponentGroupService

IIndividualGroupService extends ICompositeGroupService

IComponentGroupService. An IComponentGroupService can get and set its name and, as a component in a composite, answer its component group services. A component service can either contain other component services or be a leaf service and serve groups. While it may seem unlikely that services with groups of IPersons, like LDAP or ERP-based services would be nested inside of other service components, services with groups of IChannels actually might, particularly those representing groups of channels running on remote portals.

```
public interface IComponentGroupService {
  public Map getComponentServices();
  public Name getServiceName();
  public boolean isLeafService();
  public void setServiceName(Name newServiceName);
}
```

The reference implementation is a true composite only at service start-up, when each IComponentGroupService performs a recursive retrieval of its components. Once the elements of this composite have been retrieved, the composite service keeps its components in a one-dimensional collection. Since it does not contain nested group services, the reference composite group service does not have a direct implementation of IComponentGroupService but only implementations of its subtypes, ICompositeGroupService and IIndividualGroupService.

ICompositeGroupService. An ICompositeGroupService represents the entire composition. It is responsible for delegating requests to the appropriate component service(s) and for aggregating results. Requests come to the composite from either the outside, (the service façade), or the inside, (a component service). Some requests can be handled by a single group service, for example, a request to find a specific group (findGroup(), newGroup(), etc.) Other requests may span some or all of the component services, for example, a request to find groups that contain a particular group member (findContainingGroups()) or a request to find groups whose names contain a particular String (searchForGroups()).

```
public interface ICompositeGroupService extends IComponentGroupService
{
```

```
  public Iterator findContainingGroups(IGroupMember gm)
    throws GroupsException;
  public IEntityGroup findGroup(String key) throws GroupsException;
  public ILockableEntityGroup findGroupWithLock(String key, String owner)
    throws GroupsException;
  public IEntity getEntity(String key, Class type)
    throws GroupsException;
  public IGroupMember getGroupMember(String key, Class type)
    throws GroupsException;
  public IGroupMember getGroupMember(EntityIdentifier
    underlyingEntityIdentifier) throws GroupsException;
  public IEntityGroup newGroup(Class type, Name serviceName)
    throws GroupsException;
  public EntityIdentifier[] searchForEntities
    (String query, int method, Class type)
     throws GroupsException;
  public EntityIdentifier[] searchForEntities
    (String query, int method, Class type, IEntityGroup ancestor)
    throws GroupsException;
  public EntityIdentifier[] searchForGroups
    (String query, int method, Class leaftype)
    throws GroupsException;
  public EntityIdentifier[] searchForGroups
    (String query, int method, Class leaftype, IEntityGroup ancestor)
    throws GroupsException;
}
```

IIndividualGroupService. The third type, IIndividualGroupService defines the methods that a specific or leaf group service uses to read and write groups. IIndividualGroupService inherits find() methods from ICompositeGroupService, but whereas an ICompositeGroupService would probably delegate these requests, an IIndividualGroupService would more likely perform them itself. In addition, an IIndividualGroupService must answer if it can be updated (isEditable()), and more specifically, if it is possible to edit a particular group (isEditable(IEntityGroup group)). An attempt to update a group that is not editable should throw a GroupsException.

```
public interface IIndividualGroupService extends ICompositeGroupService
{
```

```
  public void deleteGroup(IEntityGroup group)
    throws GroupsException;
  public IEntityGroup findGroup(CompositeEntityIdentifier ent)
    throws GroupsException;
  public Iterator findMembers(IEntityGroup group)
    throws GroupsException;
  public boolean isEditable();
  public boolean isEditable(IEntityGroup group)
    throws GroupsException;
  public IEntityGroup newGroup(Class type)
    throws GroupsException;
  public void updateGroup(IEntityGroup group)
    throws GroupsException;
  public void updateGroupMembers(IEntityGroup group)
    throws GroupsException;
}
```

The reference implementation, ReferenceIndividualGroupService, delegates most requests to one of three sub-components, an IEntityGroupStore, an IEntityStore and an IEntitySearcher. It also may use the portal's Entity Locking and Entity Caching services.

Groups, Keys and Service Names

Component Services and Their Names. Once it has been assembled, the composite structure of the group service is reflected in the names given to services, which are instances of javax.naming.Name with a node for each nested service. A service named "columbia" contained by a service named "remoteChannels" would be named "remoteChannels.columbia". Since a component cannot be expected to know in advance which components will contain it, the fully-qualified service name of a given component may not be known until the composite is fully assembled. In the reference implementation, the service name is built up node by node as the composite is composed.

IEntityGroups and Their Service Names. In a composite service, the significance of service identifiers -- names -- is that they let us find the specific service that can answer a request. They also uniquely identify individual service entries, in this case groups, whose keys may not be unique across different services. Thus, a client wishing to find a group called "English101" in a component service named "ldap" needs to ask the composite service for "ldap.English101" rather than "sis.English101" or simply "English101". The ldap service may know the group as "English101" but the client must know it as "ldap.English101".

Conversely, for a service to support foreign entries, where an entry from one service participates in some way in another service, the foreign entry must be able to answer its home service identifier so that it can be retrieved, if need arises, from its service of

origin. This means that for a group to be a member of a group in another service, the member group must be able to answer its home service name. As of version 2.1, IEntityGroup inherits from org.jasig.portal.IBasicEntity, therefore a group can already answer a key and a type, in the form of an org.jasig.portal.EntityIdentifier. In the reference implementation, a group (an instance of EntityGroupImpl) answers a subclass of EntityIdentifier, CompositeEntityIdentifier, whose key contains the fully-qualified service name in addition to the native service key.

Group Members and Their Keys. From the point of view of the composite group service, the key of a group is its key in the home service concatenated to its fully-qualified service name (e.g., "ldap" + "English101" = "ldap.English101"). By contrast, the key of an entity is simply its native key, and all component group services are obligated to know it by that key. Thus, to get containing groups for IPerson "kweiner", the client would ask the service façade for an IGroupMember for IPerson "kweiner", not "ldap.kweiner" or "local.kweiner", and then ask the group member to get containing groups. The composite service would ask each of its components to get containing groups for group member IPerson kweiner, and each component service would be obligated to recognize membership information for IPerson kweiner, rather than IPerson ldap.kweiner, IPerson local.kweiner, etc. This changes our previous specification for a composite group service, but we decided to remove entity key translation from the group service api because it seemed overly burdensome and frequently unnecessary, though it may still be required internally when an ICompositeGroupService searches for entities.

Assembling the Composite

In the reference implementation, the composite service is an instance of org.jasig.portal.groups.ReferenceCompositeGroupService and is responsible for assembling the composite structure. Each leaf component is an instance of ReferenceIndividualGroupService and is customized with an IEntityGroupStore, an IEntityStore and an IEntitySearcher. A factory class for each of these types is specified in the configuration file.

The configuration file. The composite configuration is stored in xml format in properties/groups/compositeGroupServices.xml. The group service deployer edits this file to control the composition of the composite group service. The root element of this document is a service list whose service elements describe group services that are components of the top-level composite service. The configuration document is represented in Java as a GroupServiceConfiguration, essentially a parser with a Map of ComponentServiceDescriptors. Each ComponentServiceDescriptor is itself a Map containing the elements and attributes of a single service element. These elements are:

name	required
service_factory	required
entity_store_factory	required for reference implementation
group_store_factory	required for reference implementation

entity_searcher_factory	required for reference implementation
internally_managed	optional, defaults to false
caching_enabled	optional, defaults to false

Here is the service entry for the reference portal group service, which is named "local":

```
<service>
   <name>local</name>
   <service_factory>org.jasig.portal.groups.ReferenceIndividualGroupServiceFactory</service_factory>
   <entity_store_factory>org.jasig.portal.groups.ReferenceEntityStoreFactory</entity_store_factory>
   <group_store_factory>org.jasig.portal.groups.ReferenceEntityGroupStoreFactory</group_store_factory>

<entity_searcher_factory>org.jasig.portal.groups.ReferenceEntitySearcherFactory</entity_searcher_factory
>
   <internally_managed>true</internally_managed>
   <caching_enabled>true</caching_enabled>
</service>
```

Creating the component services. On service start-up, the composite service gets a GroupServiceConfiguration and asks it for its service descriptors. The composite passes each description to the appropriate service factory and gets back a new service instance.

If the component is an individual or *leaf* service, the factory creates an IIndividualGroupService, in the reference implementation, a ReferenceIndividualGroupService. The service instance uses the descriptor to customize itself, for example, by getting its group store from the group store factory designated in the descriptor. When the component has been initialized, the composite service adds the new service to its service Map.

If the service is not a leaf but a component service, the composite service asks it for its component services, which starts a recursive retrieval of leaf services. The composite service completes the naming of each leaf service by prepending the name of the top-level component to the service name, and then adds each leaf component to its service Map.

At the end of the process, non-leaf components have been eliminated, and the composite service may have multiple instances of the same IIndividualGroupService implementation, each customized by its own service descriptor.

Configuring the Composite

The configuration described in compositeGroupServices.xml is made available to group service classes via the utility class GroupServiceConfiguration. This class exposes the servicelist attributes and service elements via:

```
public Map getAttributes();
public List getServiceDescriptors()
```

uPortal 2.1 ships with the following configuration:

```
<?xml version="1.0"?>
<!-- $Revision: 1.8 $ -->
<!--
This list of component group services is processed by the composite, or "root" service as it assembles itself.
Each service element has 2 required elements: name and service_factory.  The values of all service
elements are delivered to the service_factory.
-->

<servicelist defaultService="local"
compositeFactory="org.jasig.portal.groups.ReferenceCompositeGroupServiceFactory">
  <service>
    <name>local</name>
    <service_factory>org.jasig.portal.groups.ReferenceIndividualGroupServiceFactory</service_factory>
    <entity_store_factory>org.jasig.portal.groups.ReferenceEntityStoreFactory</entity_store_factory>
    <group_store_factory>org.jasig.portal.groups.ReferenceEntityGroupStoreFactory</group_store_factory>

<entity_searcher_factory>org.jasig.portal.groups.ReferenceEntitySearcherFactory</entity_searcher_factory
>
    <internally_managed>true</internally_managed>
    <caching_enabled>true</caching_enabled>
  </service>

<!--
  <service>
    <name>ldap</name>
    <service_factory>org.jasig.portal.groups.ReferenceIndividualGroupServiceFactory</service_factory>
    <entity_store_factory>org.jasig.portal.groups.ldap.LDAPEntityStoreFactory</entity_store_factory>
    <group_store_factory>org.jasig.portal.groups.ldap.LDAPGroupStoreFactory</group_store_factory>

<entity_searcher_factory>org.jasig.portal.groups.ldap.LDAPEntitySearcherFactory</entity_searcher_factory
>
    <internally_managed>false</internally_managed>
    <caching_enabled>false</caching_enabled>
  </service>
-->

</servicelist>
```

Note that the servicelist element has 2 attributes and the "ldap" service entry is commented out.

Required servicelist Attributes. The attributes defaultService and compositeFactory are both required.

```
<servicelist defaultService="local"
 compositeFactory="org.jasig.portal.groups.ReferenceCompositeGroupServiceFactory"
```

The defaultService is the service that responds to requests for new group members when the request does not include a service name, e.g., GroupService.newGroup(Class type). The entity factory in the default service supplies those IEntities that are entry points into the composite group service (group members not obtained from other group members.) One way to substitute an alternate IEntity implementation for these entry points would be to change the default service. (Another would be to change the entity_store_factory element in the default service.) The compositeFactory attribute designates the class that creates the composite service instance. You would change its value if you wanted to substitute your own composite service implementation (and still use the configuration file.)

Additional servicelist Attributes. The GroupServiceConfiguration stores all servicelist attributes. If you wish to make additional composite service attributes available to one or

more of your component services, you can add them to the configuration document and retrieve them via:

```
String myAttribute = (String)
  GroupServiceConfiguration.getAttributes().get("myAttribute");
```

The servicelist elements. The elements of the servicelist describe *top-level* component services. The default configuration contains 2 service elements, "local", which describes the group service whose group store is the reference portal database and "ldap", which is commented out. Both of these are leaf services. The service named "local" has the following entry:

```
<service>
   <name>local</name>
   <service_factory>org.jasig.portal.groups.ReferenceIndividualGroupServiceFactory</service_factory>
   <entity_store_factory>org.jasig.portal.groups.ReferenceEntityStoreFactory</entity_store_factory>
   <group_store_factory>org.jasig.portal.groups.ReferenceEntityGroupStoreFactory</group_store_factory>

<entity_searcher_factory>org.jasig.portal.groups.ReferenceEntitySearcherFactory</entity_searcher_factory
>
   <internally_managed>true</internally_managed>
   <caching_enabled>true</caching_enabled>
</service>
```

Child Elements of the service element. Within the service element, the name and service_factory child elements are required for the assembly of the composite. In addition, the reference implementation of IIndividualGroupService requires all child elements except internally_managed and caching_enabled for a fully-functioning leaf service. The elements are as follows:
service_factory designates the class name of the factory that creates the service implementation. You only need to change this value if you are substituting your own implementation for the reference service implementation, ReferenceIndividualGroupService.
name of the service is significant if you need to use a group from the service as a composite service entry point, since you find such a group using a key that contains both the native key and the service name, e.g.,

```
String nativeKey = "100";
String serviceName = "local";
String groupKey = serviceName + "." + nativeKey;
IGroupMember myGroup = GroupService.findGroup(groupKey);
```

Warning: do not change the service name of an already-deployed group service. The service name is part of the member key in membership entries for member groups (groups that are members of other groups.) If you change it, you must change the keys of all membership entries for member groups originating in that service.

entity_store_factory is the factory class name for the entity store, the service factory for IEntities, group members that can be used to search the composite service for entity memberships. Since the keys of IEntities do not contain service names, an entity store can be shared by multiple group services. You would change this value only if you were substituting your own implementation of IEntityStore for the reference implementation.
group_store_factory is the factory class name for the group store, the adaptor that connects the group service with the native source of groups information. Although entity stores can be shared, each service will almost certainly have its own group store. The group store for "local", RDBMEntityGroupStore, refers to tables in the reference portal database, contains sql statements and retrieves group information via jdbc. The group store for "ldap", org.jasig.portal.groups.ldap.LDAPGroupStore, refers to an LDAP database and submits LDAP queries over ldap://.
entity_searcher_factory is the factory class name for the entity searcher implementation. The entity searcher is a class that refers to one or more portal entity stores and returns EntityIdentifiers for potential group members. It is likely, though by no means certain, that each group service would have its own entity searcher implementation. The entity searcher for "local" returns EntityIdentifiers for IChannels and IPersons from the reference portal database. The entity searcher for "ldap" returns EntityIdentifiers for IPersons from the same LDAP database that contains its group information.
internally_managed contains a boolean value, either *true* or *false*. If a service is internally-managed, it is under the control of the portal and presumably is capable of writing as well as reading groups. In the reference implementation, if internally_managed is *true*, the service will attempt to satisfy a request to add, update or delete a group. If it is not internally-managed, an attempt to update a group will throw a GroupsException. An alternate implementation of IIndividualGroupService could be more selective and decide if updates are allowed on a group-by-group basis.
caching_enabled has a boolean value, either *true* or *false*, which controls whether the component service uses the portal's entity caching service to cache groups. The value for the "local" service is set to *true* to eliminate excess database calls. It is set to *false* for "ldap" because the ldap service has its own caching mechanism.
The ldap service. Comment in the service element named "ldap" if you want to hook up an ldap group service:

```
  <service>
    <name>ldap</name>
    <service_factory>org.jasig.portal.groups.ReferenceIndividualGroupServiceFactory</service_factory>
    <entity_store_factory>org.jasig.portal.groups.ldap.LDAPEntityStoreFactory</entity_store_factory>
    <group_store_factory>org.jasig.portal.groups.ldap.LDAPGroupStoreFactory</group_store_factory>

<entity_searcher_factory>org.jasig.portal.groups.ldap.LDAPEntitySearcherFactory</entity_searcher_factory
>
    <internally_managed>false</internally_managed>
    <caching_enabled>false</caching_enabled>
  </service>
```

Note that the value of service_factory is the same for both the "local" and "ldap" service elements. This is because both the local and ldap group services are instances of

ReferenceIndividualGroupService, customized by their respective service configurations. The entries for entity_store_factory, group_store_factory and entity_searcher_factory all designate different factory classes. Nonetheless, they all return the same (singleton) instance of org.jasig.portal.groups.ldap.LDAPGroupStore, which implements the IEntityStore, IEntityGroupStore and IEntitySearcher interfaces. The internally_managed element is set to *false* because in most environments, the LDAP store will not be updatable by the portal and in any event, LDAPGroupStore does not support updates.

The LDAP Group Store

The LDAP group store gets its entity memberships from LDAP and its group structure from a configuration file, properties/groups/LDAPGroupStoreConfig.xml. The configuration file also defines the location of the native group store. (The dtd for LDAPGroupStoreConfig.xml is LDAPGroupStore.dtd.) Understanding the configuration file is crucial to understanding how the group store works.

Configuration file elements. The configuration file has a single config element that describes the LDAP connection and some number of group elements that establish the groups structure and ultimately, map LDAP queries to groups. The config element currently points to the Columbia University public LDAP server:

```
<config>
   <url>ldap://ldap.columbia.edu:389/o=Columbia%20University,c=US</url>
   <logonid></logonid>
   <logonpassword></logonpassword>
   <keyfield>uni</keyfield>
   <namefield>cn</namefield>
   <usercontext></usercontext>
   <refresh-minutes>120</refresh-minutes>
</config>
```

Group elements. The group element contains the IEntityGroup attributes key, name and description. Just as an IEntityGroup can contain groups and entities, a group element can contain other member group elements, and it can contain member entities in the form of a single entity-set. At its simplest, an entity-set can contain a filter element that defines an LDAP query that returns entities. Or it may contain a union, intersection, difference or subtract element, which, in turn, contains entity-sets whose results are combined through one of the following operations:

union	all entities from all contained entity-sets
intersection	all common entities from all contained entity-sets
difference	all entities that occur in only one contained entity-set
subtract	all entities in the first entity-set minus any that occur in the rest

The Group Samples. The samples provide examples of each of these entity-set types. The configuration file ships with 7 sample groups, described below. They form the following structure (the group key is in parentheses):

All LDAP Groups (all)
  Vigdors(1)
  Fracapanes and Ellentucks (2)
  Union test (3)
  Intersection test (4)
  Difference test (5)
  Subtract test (6)

- *All LDAP Groups (all)* contains groups "1" thru "6" but contains no member entities. It does not (directly) pull information from LDAP.
- *Vigdors (1)* contains an entity-set with a filter element whose String value is cn=*vigdor. The members of *Vigdors* are the entities returned by this query.
- *Fracapanes and Ellentucks (2)* contains an entity-set with a union element that *or's* the results of two entity-set elements, each of which has a filter element describing an LDAP query. The resulting entity-set contains those entities returned by LDAP from the query cn=*fracapane plus those entities returned by cn=*ellentuck.
- *Union test (3)* is similar to *Fracapanes and Ellentucks (2)* in that it contains an entity-set with a union element containing 2 entity-sets each of which has a filter element. The resulting entity-set contains the union of cn=donald f* and cn=*frac*.
- *Intersection test (4)* contains an entity-set with an intersection element that *and's* the results of two entity-set elements, each of which has a filter element. The resulting entity-set contains the intersection of cn=donald f* and cn=*frac*.
- *Difference test (5)* contains an entity-set with a difference element that *exclusive-or's* the results of two entity-set elements, each of which has a filter element. The resulting entity-set contains the entities returned by one but not both of cn=donald f* and cn=*frac*.
- *Subtract test (6)* contains an entity-set with a subtract element that *subtracts* the results of one entity-set from another. Each entity-set has a filter element, and the resulting entity-set contains the entities returned by cn=donald f* but not by cn=*frac*.

Limitations of the LDAP Group Store. It is important to understand what the LDAPGroupStore does and does not do and why it does not support updates. The store queries LDAP to discover entities that are group members, but it looks at its configuration file to discover groups and their relationships. As a result, the store will discover an entity added to LDAP (provided it is returned by an entity-set defined in the configuration file.) But it will not discover any group beyond those defined in the configuration file. The way to add a group to the store is to add a group element to the configuration document. Likewise, the way to make a group a member of another group in the store is to add a group element to another group element in the configuration document. On the other hand, the way to add an entity to a group from the store is to add or update the entity in LDAP. The LDAP group store could support updates to the group structure (adding or deleting member groups) if it had the ability to update the configuration document. It could support updates to entity memberships if it had the ability to update LDAP.

Deploying the LDAP Group Store Locally. The configuration file as delivered points to the Columbia University public LDAP server and defines a few not-terribly-useful groups. To implement a local LDAP group service that uses the LDAPGroupStore, modify the configuration file so that the config element points to your LDAP server. Then, replace the entity-sets and filters with queries that return meaningful results in your environment. When you do this, give your groups appropriate names and make sure their keys are unique. For example, you might convert the structure into something like:

All LDAP Groups (all)
- Faculty Groups (1)
  - Biology Department (3)
  - Chemistry Department (4)
- Portal Staff (2)
  - Portal Administrators (5)
  - Portal Developers (6)

Now un-comment the "ldap" service element in compositeGroupServices.xml and start up the portal. In the *Groups Manager* channel, try adding *All LDAP Groups* to the root group *Everyone*. (For instructions on using *Groups Manager*, see The Groups Manager Channel.) The groups that you defined in your LDAP configuration file should now be available for browsing in Groups Manager, although you won't be able to update them. Once you are comfortable with the process of defining groups in the configuration file, you can begin the task of deriving a group structure from LDAP that includes your portal population and models your organization.

Next Steps

The process of deploying a composite group service involves (at least) the following steps:

- analyze why you need group information
- identify the necessary sources of group information
- find or create adaptors for these sources
- configure the composite service

Most portals will at least use groups to manage authorization, so this is a common starting point. Many institutions will rely on an LDAP service as the primary source of group information and supplement it with information from the portal database. Others may require additional information from human resources, student information or other systems. uPortal currently ships with 2 adaptors, 1 for LDAP (org.jasig.portal.groups.ldap.LDAPGroupStore) and 1 for the reference portal database (RDBMEntityGroupStore). If you only intend to use these 2 sources, you do not have to write a custom group store. If you do need to draw groups from another source, you will have to implement the IEntityGroupStore and IEntitySearcher interfaces. You should not have to write a custom group serivce (an IIndividualGroupService) unless you need to change the transactional rules of the service. Of course, you can always re-implement or sublcass the reference implementations for reasons of efficiency, correctness or to add new functions. Please contribute your code back to the project so that the entire uPortal community can benefit from your improvements.

The final step is to represent your composite group service in the configuration file. Describe each top-level service in a service element and designate the default service. Unless you have a specific reason for changing it, keep the initial default value of "local". If a service supports updates, set internally_managed to *true*. Note that you must implement the update methods for such a service in any custom group store class. Caching of groups is a must in a production system. Either use the portal Entity Caching Service or implement your own caching mechanism.

Please post your questions, comments, suggestions and ideas to the JA-SIG Portal Discussion List. Good luck!

de 3/4/03