ApacheDS Bootstrapping

Introduction

This document describes the process where ApacheDS is first started up, starts to initialize various components, to finally reach a solid state of operation where it can service LDAP request along with other directory backed protocol requests.

We break up this process into several different categories as phases in the bootstrapping process, since each phase involves a different set of facilities and components. The phases are:

1. Daemon Bootstrapping Phase
2. Initial Bootstrap Schema Load Phase
3. Full Schema Load Phase
4. System/Configuration Partition Initialization Phase
5. Wiring and DI Phase
6. Startup Phase

Daemon Bootstrapping Phase

Several daemon frameworks have been used in the past to make ApacheDS' process act like a daemon on *NIX platforms and as a service on Windows platforms. These frameworks were Procrun, Jsvc, and the Tanuki wrapper.

Each framework has a means to start up a Java program and set it up so it detaches from the console and becomes a daemon on UNIX and/or a service on Windows. How this is done is abstracted away by some minimal code in ApacheDS used to bootstrap it. Some of this code is very specific to the wrapper and some is quit general.

Bootstrappers

In ApacheDS' repository there's a daemon-bootstrappers project which contains a base Bootstrapper implementation and some helper classes along with various framework subtypes of the Bootstrapper.

The goal here was to adapt the set of events handled by various frameworks to startup a Java based daemon, to a generalized interface. In this case the generalized interface is really a class called Bootstrapper.

Tanuki Bootstrapper

The Tanuki framework has a TanukiBootstrapper implementation extending the Bootstrapper class. On startup the Tanuki framework calls the main(String []) method of this TanukiBootstrapper. The apacheds.conf file contains configuration entries telling Tanuki to start ApacheDS by calling this class' main method with various arguments.

The static main() method in the TanukiBootstrapper simply instantiates a new TanukiBootstrapper instance and asks the wrapper framework (WrapperManager) to start up the TanukiBootstrapper. Here are the String[] args handed off to the main() method:

```
/usr/local/apacheds-1.5.5-SNAPSHOT,
org.apache.directory.server.Service,
/usr/local/apacheds-1.5.5-SNAPSHOT/instances/default/conf/server.xml
```

Future Work

We envision a configuration free of Maven plugins to generate the installers. We also would like to modify the InstallationLayout bean and the InstanceLayout bean so that the InstanceLayout bean has a handle on the InstallationLayout. The InstanceLayout bean will be provided to higher layers for the server to properly conduct it's bootstrap process in later phases.

Initial Bootstrap Schema Load Phase

We store the schema in a partition. We also store configuration state information regarding which schema is to be enabled and which schema are disabled in this partition. Everything that is needed for loading and setting up the schema subsystem is located in this partition.

Also this partition can use any kind of backing store, or Partition interface implementation so those wishing to store schema information in various backing stores can do so.

Pre-packaged Schema LDIF Files

The ldap-schema project in shared produces a jar file which contains LDIF files within in a specific layout. These LDIF files encode schema information for various published schema as well as a minimal set of critical schema specifically required for ApacheDS operation.

The layout in the jar file contains a top level **schema** directory. Under this there is a schema.ldif file and yet another **schema** directory. This ldif and the directory is analogous to the ou=schema context in ApacheDS.

Under this second **schema** directory you'll find a set of directories with the names of various schema in lower case:

- apache
- apachedns
- apachemeta
- autofs
- collective
- corba
- core
- cosine
- dhcp
- inetorgperson
- java
- krb5kdc
- mozilla
- nis
- other
- samba
- system

Along side each of these schema directories rests an LDIF file with the same name plus the **.ldif** extension. So next to the **java** directory you'll find **java.ldif**.

Under each schema directory there is a standard structure for storing the various kinds of schema entities: syntaxCheckers, syntaxes, comparators, normalizers, matchingRules, attributeTypes, objectClasses, nameForms, matchingRuleUses, ditStructureRules, and ditContentRules.

For each kind of schema entity you have a directory with the name of the entities above. Like for example there is a **schema/schema/java/attributeTypes** directory in the java schema for storing LDIF files of each attributeType in this schema. For the same name of the this directory you have an LDIF file beside it: **schema/schema/java/attributeTypes.ldif**. Inside each schema entity directory you'll have LDIF files. The LDIF file will use the OID of the schema object as the base file name and the standard LDIF file extension. So for example in the java schema you will have this file: **schema/schema/java/attributeTypes/1.3.6.1.4.1.42.2.27.4.1.10.ldif**. The file for example contains the following LDIF:

```
dn: m-oid=1.3.6.1.4.1.42.2.27.4.1.10,ou=attributeTypes,cn=java,ou=schema
m-equality: caseExactMatch
m-usage: USER_APPLICATIONS
objectclass: metaAttributeType
objectclass: metaTop
objectclass: top
createtimestamp: 20090818022732Z
creatorsname: uid=admin,ou=system
m-name: javaFactory
m-oid: 1.3.6.1.4.1.42.2.27.4.1.10
m-singlevalue: TRUE
m-description: Fully qualified Java class name of a JNDI object factory
entryuuid:: bVjvv73vv70Q77+9Q8ySWmNj77+977+9dwM=
m-collective: FALSE
m-obsolete: FALSE
m-nousermodification: FALSE
entrycsn: 20090818052732.339000Z#000000#000#000000
m-syntax: 1.3.6.1.4.1.1466.115.121.1.15
```

The schema used to encode the schema itself is called the apachemeta schema. These m- attributes and the metaXXX values you see for objectClass all come from the apachemeta schema. So as you can see to properly load and make sense of this data you at first need to know the apachemeta schema and this presents a slight chicken and egg problem for us that we must overcome.

So this apacheds-ldap-schema-[version].jar file has all these LDIF files zipped up inside it. In addition it contains a special extractor which will extract out this zipped schema repository onto disk.

Solving the Chicken and Egg Problem

As we stated earlier we have a slight chicken and egg issue to overcome. To load the schema into registries from a partition containing this information we have to be aware at least of the apachemeta schema and the other schemas it depends on. This dependencies pull in the following schema as the critical set of schemas required to read and bootstrap all schema:

- apache
- apachemeta
- core
- system

Let's call this critical schema set, the **bootstrap schema set**. We need to load a set of registries with this schema set in order to initialize a partition containing entries storing all the schema information to be loaded. Remember our server is designed to store schema in a partition which is accessible via LDAP in this well organized structure in addition to using the standard LDAP mechanism which uses a schemaSubentry.

So how do we do this? These 4 schema are so critical that they are read only. No administrator should ever be allowed to change the elements in the bootstrap schema set. For this reason we are comfortable loading these critical elements from the ldap-schema jar without extracting them to disk. We also read them without having active registries.

Without active registries schema checking is not possible. Nor is schema integrity checks possible which make sure the schema is right before loading it. Instead we just load and hope all goes well in the end with a post load integrity check. This however is not a major issue if no one is allowed to change the bootstrap schema set.

So the initial bootstrap schema loading stage loads these 4 critical schema into the registries. This is done by a special Partition implementation called SchemaPartition. The SchemaPartition uses a special JarLdifSchemaLoader class to load these four schema into a Registries object.

The SchemaPartition itself does not store schema data in some persistent store. Instead this Partition delegates persistence to a wrapped Partition that it contains. The idea here is to allow anyone to replace the underlying storage used to store schema data in ApacheDS. Hence any Partition implementation such as the, OraclePartition, JdbmPartition, or LdifPartition can be plugged in.

Regardless of the Partition implementation used, the partition must be self configuring. Meaning it must not require an external configuration to come online and provide access to the schema data it stores. This is because when this partition is brought online, the configuration partition of ApacheDS is not yet available.

Full Schema Load Phase

The SchemaPartition uses the initial critical set of bootstrap schema to initialize it's wrapped Partition, whatever implementation it may be. The SchemaPartition delegates Partition interface operations to this wrapped Partition while also updating the registries to reflect the changes to schema during solid state operation of the server.

Once the wrapped Partition with our schema content in the expected DIT namespace is initialized the second full schema load phase can begin. In this phase, a second schema loader is incorporated to load the schema from within a Partition. This loader uses the Partition interface methods to search the Partition and load the Registries with all the schema enabled in this partition. For example in the **ou=schema,cn=java** entry you'll find the same content that was bundled into the ldap-schema jar LDIF **schema/schema/java.ldif**:

```
dn: cn=java,ou=schema
entryuuid:: 77+9Fxdi77+9e07vv73vv73vv73vv73vv70g77+9aFQ=
creatorsname: uid=admin,ou=system
createtimestamp: 20090818022726Z
cn: java
objectclass: metaSchema
objectclass: top
entrycsn: 20090818052726.390000Z#000000#000#000000
m-dependencies: system
m-dependencies: core
m-disabled: FALSE
```

NOTE the m-disable attribute is set to FALSE. This LDIF or entry in the wrapped Partition tells the PartitionSchemaLoader that it must load the Java schema into the registries in this full schema load phase. The first 4 schema in the critical bootstrap schema set do not need to be reloaded since they are already enabled and present in the Registries object due to the initial bootstrap schema load phase.

System/Configuration Partition Initialization Phase

This phase involves initializing the system partition which stores the configuration for the server and it's various components and subsystems. The configuration is stored in a partition whose DIT structure is exposed and managed via LDAP. This is what we call CiDIT; it is short for configuration in DIT.

Now that we have all the schema that is packaged with the system or previously custom installed by the user on previous runs, loaded, we can start up all the other partitions.

Don't we know what schema is used in the system partition?

The short answer is NO!

The long answer requires knowledge of how we plan to expose extension points in the server for various kinds of components along with their configuration information. The most familiar extension point is the ability to plug in any kind of Partition implementation into the server. Not every Partition implementation will have the same configuration parameters. Each implementation might want to expose a specific schema to represent it's configuration parameters in the configuration DIT. For this reason we will never know the bounds of the schema utilized in the system schema and must load the entire schema stored within the schema partition to initialize the system partition storing the configuration information for the server.

To understand this full read on into the next phase of the bootstrapping process, the wiring and dependency injection phase.

Wiring and DI Phase

Once the configuration is online through the system partition, we have all the information we need to configure all the components within the server. The process of instantiating, configuring, and wiring together components to assemble a fully functional server instance that is ready to start is managed by this phase of the bootstrapping process.

Right now there is heavy discussion on what we should use to do this. Obviously this topic is vast since frameworks like Spring and OSGi have been devised specifically for this purpose. We suspect that at the end of the day we will depend on OSGi since this is the preferred framework of choice on our team.

Today we need an intermediate solution to cope with the transition from Spring to OSGi. However before we start discussing this and how to get to the end state let's talk about the nirvana we would like to reach.

Reaching Component Nirvana a la LDAP

Just imagine writing or installing a 3rd party component implementation for an ApacheDS extension point like say a Partition. This component would be an OSGi module bundled with LDIF schema resources.

Using studio or an LDAP client/tool, the user installs the component bundle via LDAP onto the server. The installation process creates an entry within the component registry of ApacheDS in the configuration area of the DIT. This component registry entry simply encodes the OSGi MANIFEST entries in the bundle as an LDIF and adds additional information that could be used to manage the bundle within ApacheDS. Next the packaged schema LDIF files are added to the schema partition under an existing schema or a new one depending on how these files are laid out in the jar. The schema is then enabled and loaded into the Registries. The actual physical jar bundle is placed in a bundle repository area on disk of the ApacheDS instance.

Then under the configuration area for server partitions, an administrator can just create a new entry using the objectClass associated with the component. This entry would contain values for all the configuration parameters associated with the component. The addition of the configuration entry can trigger the instantiation and initialization of the component, and the partition can come online. However this is not a good idea. Because sometimes a component will have nested configuration parameters which may be modeled as nested entries, it is important to prevent triggering of a component load before a consistent configuration is stored in the configuration area. To prevent this problem, an enable/disable bit can be used. So before the entire configuration is completely added, the disabled flag can be set to TRUE. Then when the component configuration is consistent the disabled attribute in the entry can be deleted or set to FALSE. This then can trigger the start up of the component instance using the supplied configuration in the DIT.

How do we get to Component Nirvana from Spring Hell?

Well the best thing to do immediately is to start using interfaces to describe services that can plug into an OSGi container. Then implementations can be managed as bundles in separate modules.

The next couple steps need to be taken together in one shot because of how interdependent they are. We need to define the component registry, and the component configuration instance are with specific schema elements to begin describing the different kinds of extension points we have. Once these are defined the existing implementations need to be turned into bundles and added to a default registry that comes prepackaged with the server. The same goes for the initial instance configurations for these components. This is all relatively easy to do.

The next step is to build a component wiring module that leverages an existing OSGi framework implementation, (eehhem Felix) to wire together the various components while configuring them. Felix does most of this for us through the OSGi service dependency management mechanisms it supports.

Now we're going to need to build a neat little implementation of an OSGi service called the ConfigAdmin. As the name suggests, we will use this implementation to pull configuration information from the CiDIT area for various instances, to configure the services. Besides reading this information the ApacheDSConfigAdmin will also notify components of potential re-configuration events when administrators use LDAP to change the configurations of component instances in the CiDIT area or upgrade component bundles to new releases.

This dynamic reconfiguration mechanism will allow ApacheDS to run uninterrupted while reconfiguring and updating itself and its components. Lofty idea, and whether this will work or not in complex situations we don't know. If it can work for minor tweaks we should be happy.

Startup Phase

Once all the components are wired together and the server is ready to service requests, the final steps for toggling the online mode will be taken. This may be as simple as flipping some binary flag to open the flood gates or just binding to the service ports.