KIP-8 Service Discovery and Topology Generation

Status

Current state: *Complete*

Discussion thread:

JIRA: KNOX-1006

Motivation

There are a number of usability issues that are related to the current need to edit possibly large XML topology files within the Apache Knox Admin UI and within

Apache Ambari. Some of these issues are:

1. The hierarchical structure of our provider configuration
2. Having to specify the correct URL of the services being proxied
3. Redundantly configuring the same LDAP configuration across different topologies
4. Redundantly configuring and picking up changes to cluster configuration such as HA enablement for one or more proxied services
5. Difficulty in providing topology management within Ambari given the typical name value pair style of Ambari

By providing a Service Discovery service in Knox, we can provide a more dynamic mechanism for generating, populating or realizing service details such as URL, HA enablement, etc.

By introducing a greatly simplified descriptor that can be combined with service discovery at deployment time, we can provide much simpler user experience in Ambari and the Knox Admin UI.

This KIP will propose an approach to achieving the above.

Service Discovery

1. Service Registry

As some background information, in the early days of Apache Knox, we had the idea of a service registry as part of the gateway itself.

The idea was that as services spun up they would acquire a registrationCode and sign it for registration.

Service URLs could then be looked up for each "cluster" (topology) from this global registry.

There is an internal representation of this registry in Knox now and it is populated today by the deployment machinery - signed registration code and all.

The registry is serialized to the {GATEWAY_HOME}/data/security/registry file.

It is later deserialized on restart of the gateway in order to have the in-memory model needed by the gateway but since the deployment machinery doesn't run on restart for existing topologies we need to preserve the previous state.

There are a couple things that aren't great about this mechanism:

1. The fact that it is tied to topologies initial deployment necessitates the serialized registry. If we can remove this file it would be very good. It introduces backward compatibility challenges at times.
2. The fact that this gateway service is a global namespace for topology/services means that the management of the central registry is more tedious than it would be if each topology has its own specific namespace that came and went with the topology itself.
3. The whole signed registrationCode is not needed since when Knox is being its own registry it doesn't really need authentication to be done for each

This whole thing may be able to be eliminated or replaced by Service Discovery or may need to continue to be supported with some improvements for deployments that do not have another registry like Ambari available.

We will need to make sure that we continue to meet the needs that this existing mechanism provides.

2. Simplified Topology Descriptor

In order to simplify what management UI's need to manage, we should support a much simpler descriptor that contains what is needed to generate an actual topology file at deployment time (see item #1 from the Motivation section).

A simplified topology descriptor must allow the following details to be specified:

- Service discovery type
  An identifier indicating which type of discovery to apply (e.g., Ambari, etc...)

- Service discovery address
  The associated service registry address
- Credentials for interacting with the discovery source
- A provider configuration reference (a unique name, filename, etc...)
  A unique name mapped to a set of provider configurations (see item #3 from the Motivation section)
- A list of services to be exposed through Knox (with optional service parameters and URL values)
- A list of UIs to be proxied by Knox (per KIP-9)

YAML offers more structure than a properties file, and is suitable for adminstrators/developers hand-editing simple descriptors.

Proposed YAML

```
# Discovery info source
discovery-type: AMBARI
discovery-address: http://sandbox.hortonworks.com:8080
discovery-user: maria_dev
discovery-pwd-alias: ambari.discovery.password

# Provider config reference, the contents of which will be
# included in (or referenced from) the resulting topology descriptor.
# The contents of this reference has a <gateway/> root, and
# contains <provider/> configurations.
provider-config-ref : sandbox-providers.xml

# The cluster for which the service details should be discovered
cluster: Sandbox

# The services to declare in the resulting topology descriptor,
# whose URLs will be discovered (unless a value is specified)
services:
    - name: NAMENODE
    - name: JOBTRACKER
    - name: WEBHDFS
    - name: WEBHCAT
    - name: OOZIE
    - name: WEBHBASE
    - name: HIVE
    - name: RESOURCEMANAGER
    - name: KNOXSSO
      params:
          knoxsso.cookie.secure.only: true
          knoxsso.token.ttl: 100000
    - name: AMBARI
      urls:
        - http://sandbox.hortonworks.com:8080
    - name: AMBARIUI
      urls:
        - http://sandbox.hortonworks.com:8080

# UIs to be proxied through the resulting Knox topology (see KIP-9)
#uis:
#   - name: AMBARIUI
#     url: http://sandbox.hortonworks.com:8080
```

While JSON is not really a format for configuration, it is certainly appropriate as a wire format, and will be used for API interactions.

Proposed JSON

```
{
  "discovery-type":"AMBARI",
  "discovery-address":"http://sandbox.hortonworks.com:8080",
  "discovery-user":"maria_dev",
  "discovery-pwd-alias":"ambari.discovery.password",
  "provider-config-ref":"sandbox-providers.xml",
  "cluster":"Sandbox",
  "services":[
     {"name":"NAMENODE"},
     {"name":"JOBTRACKER"},
     {"name":"WEBHDFS"},
     {"name":"WEBHCAT"},
     {"name":"OOZIE"},
     {"name":"WEBHBASE"},
     {"name":"HIVE"},
     {"name":"RESOURCEMANAGER"},
     {"name":"KNOXSSO",
      "params":{
          "knoxsso.cookie.secure.only":"true",
          "knoxsso.token.ttl":"100000"
      }
     },
     {"name":"AMBARI", "urls":["http://sandbox.hortonworks.com:8080"]}
  ],
  "uis":[
     {"name":"AMBARIUI", "urls":["http://sandbox.hortonworks.com:8080"]}
  ]
}
```

3.Topology Generation

Given that we will have a Service Discovery service that can integrate with Ambari as well as other sources of needed metadata, we should be able to start with a simplified topology descriptor.
Once the deployment machinery notices this descriptor, it can pull in the referenced provider configuration, iterate over each of the services, UIs, applications and lookup the details for each.
With the provider configuration and service details we can then generate a fully baked topology.

From the example descriptors in the Simplified Topology Descriptors section, the resulting topology should include the provider configuration from the reference, and each service and UI (another KIP) with its respective URLs:

Sample Topology File

```
<?xml version="1.0" encoding="UTF-8"?>
<topology>
    <gateway>
        <provider>
            <role>authentication</role>
            <name>ShiroProvider</name>
            <enabled>true</enabled>
            <param>
                <!--
                session timeout in minutes,  this is really idle timeout,
                defaults to 30mins, if the property value is not defined,,
                current client authentication would expire if client idles contiuosly for more than this value
                -->
                <name>sessionTimeout</name>
                <value>30</value>
            </param>
            <param>
                <name>main.ldapRealm</name>
                <value>org.apache.hadoop.gateway.shirorealm.KnoxLdapRealm</value>
```

```
        </param>
        <param>
            <name>main.ldapContextFactory</name>
            <value>org.apache.hadoop.gateway.shirorealm.KnoxLdapContextFactory</value>
        </param>
        <param>
            <name>main.ldapRealm.contextFactory</name>
            <value>$ldapContextFactory</value>
        </param>
        <param>
            <name>main.ldapRealm.userDnTemplate</name>
            <value>uid={0},ou=people,dc=hadoop,dc=apache,dc=org</value>
        </param>
        <param>
            <name>main.ldapRealm.contextFactory.url</name>
            <value>ldap://localhost:33389</value>
        </param>
        <param>
            <name>main.ldapRealm.contextFactory.authenticationMechanism</name>
            <value>simple</value>
        </param>
        <param>
            <name>urls./**</name>
            <value>authcBasic</value>
        </param>
    </provider>
    <provider>
        <role>identity-assertion</role>
        <name>Default</name>
        <enabled>true</enabled>
    </provider>
    <!--
    Defines rules for mapping host names internal to a Hadoop cluster to externally accessible host names.
    For example, a hadoop service running in AWS may return a response that includes URLs containing the
    some AWS internal host name.  If the client needs to make a subsequent request to the host identified
    in those URLs they need to be mapped to external host names that the client Knox can use to connect.
    If the external hostname and internal host names are same turn of this provider by setting the value of
    enabled parameter as false.
    The name parameter specifies the external host names in a comma separated list.
    The value parameter specifies corresponding internal host names in a comma separated list.
    Note that when you are using Sandbox, the external hostname needs to be localhost, as seen in out
    of box sandbox.xml.  This is because Sandbox uses port mapping to allow clients to connect to the
    Hadoop services using localhost.  In real clusters, external host names would almost never be localhost.
    -->
    <provider>
        <role>hostmap</role>
        <name>static</name>
        <enabled>true</enabled>
        <param><name>localhost</name><value>sandbox,sandbox.hortonworks.com</value></param>
    </provider>
</gateway>

<service>
    <role>AMBARIUI</role>
    <url>http://c6401.ambari.apache.org:8080</url>
</service>
<service>
    <role>HIVE</role>
    <url>http://c6402.ambari.apache.org:10001/cliservice</url>
</service>
<service>
    <role>WEBHCAT</role>
    <url>http://c6402.ambari.apache.org:50111/templeton</url>
</service>
<service>
    <role>AMBARI</role>
    <url>http://c6401.ambari.apache.org:8080</url>
</service>
<service>
    <role>OOZIE</role>
    <url>http://c6402.ambari.apache.org:11000/oozie</url>
```

```
    </service>
    <service>
        <role>JOBTRACKER</role>
        <url>rpc://c6402.ambari.apache.org:8050</url>
    </service>
    <service>
        <role>NAMENODE</role>
        <url>hdfs://c6401.ambari.apache.org:8020</url>
    </service>
    <service>
        <role>WEBHBASE</role>
        <url>http://c6401.ambari.apache.org:60080</url>
    </service>
    <service>
        <role>WEBHDFS</role>
        <url>http://c6401.ambari.apache.org:50070/webhdfs</url>
    </service>
    <service>
        <role>RESOURCEMANAGER</role>
        <url>http://c6402.ambari.apache.org:8088/ws</url>
    </service>
    <service>
        <role>KNOXSSO</role>
        <param>
            <name>knoxsso.cookie.secure.only</name>
            <value>true</value>
        </param>
        <param>
            <name>knoxsso.token.ttl</name>
            <value>100000</value>
        </param>
    </service>
</topology>
```

3.1 Simple Descriptor Discovery

We should also consider how we will discover simple descriptors and I think that we may want to have multiple ways.

3.1.1 Local

Just as is currently done for topology files, Knox can monitor a local directory for new or changed descriptors, and trigger topology generation and deployment upon such events.
This is great for development and small cluster deployments.

The Knox Topology Service will monitor two additional directories:

- **conf/shared-providers**
  - Referenced provider configurations will go in this directory; These configurations are the <gateway/> elements found in topology files.
  - When a file is modified (create/update) in this directory, any descriptors that reference it are updated to trigger topology regeneration to reflect any provider configuration changes.
  - Attempts to delete a file from this directory via the admin API will be prevented if it is referenced by any descriptors in conf/descriptors.

- **conf/descriptors**
  - Simple descriptors will go in this directory.
  - When a file is modified (create/update) in this directory, a topology file is (re)generated in the conf/topologies directory.
  - When a file is deleted from this directory, the associated topology file in conf/topologies is also deleted, and that topology is undeployed.
  - When a file is deleted from the conf/topologies directory, the associated descriptor in conf/descriptors is also deleted (if it exists), to prevent unintentional regeneration/redeployment of the topology.

3.1.2 Remote

For production and larger deployments, we need to be able to accommodate multiple instances of Knox better. One proposal for such cases is a ZooKeeper-based discovery mechanism.
All Knox instances will pick up the changes from ZK as the central source of truth, and perform the necessary generation and deployment of the corresponding topology.

The location of these descriptors and their dependencies (e.g., referenced provider config) in ZK must be defined.

It would also be helpful to provide a means (e.g., Ambari, Knox admin UI, CLI, etc...) by which these descriptors can be easily published to the correct location in a znode.

4. Service Discovery

Item #2 from the Motivation section addresses the usability issues associated with having to manually edit a topology descriptor to specify the various service URLs in a cluster.
Beyond the obvious tedium associated with this task, there is real potential for human error, especially when multiple Knox instances are involved. The proposed solution to this
is automated service discovery, by which the URLs for the services to be exposed are discovered and added to a topology.

In order to transform a Simplified Topology Descriptor into a full topology, we need some source for the relevant details and metadata. Knox is often deployed in clusters
with Ambari and ZooKeeper also deployed, and both contain some level of the needed metadata about the service URLs wtihin the cluster.

Ambari provides everything we need to flesh out a full topology from a minimal description of what resources that we want exposed and how we want access to those resources protected.

The ZooKeeper service registry has promise, but we need to investigate the level of use from a registered services perspective and details available for each service.

4.1 Apache Ambari API

These are some **excerpts** of responses from the Ambari REST API which can be employed for topology discovery:

(*CLUSTER_NAME* is a placeholder for an actual cluster name)

Identifying Clusters - /api/v1/clusters

/api/v1/clusters

```
{
  "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/",
  "items" : [
    {
      "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/CLUSTER_NAME",
      "Clusters" : {
        "cluster_name" : "CLUSTER_NAME",
        "version" : "HDP-2.6"
      }
        }
  ]
}
```

Service Component To Host Mapping - /api/v1/clusters/CLUSTER_NAME/services?fields=components/host_components/HostRoles

/api/v1/clusters/CLUSTER_NAME/services?fields=components/host_components/HostRoles

```
“items” : [
  {
    “href” : "http://AMBARI_ADDRESS/api/v1/CLUSTER_NAME/CLUSTER_NAME/services/HIVE",
    “components” : [
      {
        “ServiceComponentInfo” : { },
        “host_components” : [
          {
            “HostRoles” : {
              “cluster_name” : “CLUSTER_NAME”,
              “component_name” :  “HCAT”,
              “host_name” : “c6402.ambari.apache.org”
            }
          }
        ]
      },
      {
        “ServiceComponentInfo” : { },
        “host_components” : [
          {
            “HostRoles” : {
              “cluster_name” : “CLUSTER_NAME”,
              “component_name” :  “HIVE_SERVER”,
              “host_name” : “c6402.ambari.apache.org”
            }
          }
        ]
      }
        ]
  },
  {
    “href” : "http://AMBARI_ADDRESS/api/v1/CLUSTER_NAME/CLUSTER_NAME/services/HDFS",
    “ServiceInfo” : {},
    “components” : [
      {
        “ServiceComponentInfo” : { },
        “host_components” : [
          {
            “HostRoles” : {
              “cluster_name” : “CLUSTER_NAME”,
              “component_name” :  “NAMENODE”,
              “host_name” : “c6401.ambari.apache.org”
            }
          }
        ]
      }
        ]
  }
]
```

Service Configuration Details (Active Versions) - /api/v1/clusters/ CLUSTER_NAME /configurations/service_config_versions?is_current=true

/api/v1/clusters/ CLUSTER_NAME /configurations/service_config_versions?is_current=true

```
"items" : [
  {
      "href" : "http://AMBARI_ADDRESS/api/v1/clusters/CLUSTER_NAME/configurations/service_config_versions?
service_name=HDFS&service_config_version=2",
      "cluster_name" : "CLUSTER_NAME",
      "configurations" : [
        {
          "Config" : {
            "cluster_name" : "CLUSTER_NAME",
            "stack_id" : "HDP-2.6"
```

```
        },
        "type" : "ssl-server",
        "properties" : {
          "ssl.server.keystore.location" : "/etc/security/serverKeys/keystore.jks",
          "ssl.server.keystore.password" : "SECRET:ssl-server:1:ssl.server.keystore.password",
          "ssl.server.keystore.type" : "jks",
          "ssl.server.truststore.location" : "/etc/security/serverKeys/all.jks",
          "ssl.server.truststore.password" : "SECRET:ssl-server:1:ssl.server.truststore.password"
        },
        "properties_attributes" : { }
      },
      {
        "Config" : {
          "cluster_name" : "CLUSTER_NAME",
          "stack_id" : "HDP-2.6"
        },
        "type" : "hdfs-site",
        "tag" : "version1",
        "version" : 1,
        "properties" : {
          "dfs.cluster.administrators" : " hdfs",
          "dfs.encrypt.data.transfer.cipher.suites" : "AES/CTR/NoPadding",
          "dfs.hosts.exclude" : "/etc/hadoop/conf/dfs.exclude",
          "dfs.http.policy" : "HTTP_ONLY",
          "dfs.https.port" : "50470",
          "dfs.journalnode.http-address" : "0.0.0.0:8480",
          "dfs.journalnode.https-address" : "0.0.0.0:8481",
          "dfs.namenode.http-address" : "c6401.ambari.apache.org:50070",
          "dfs.namenode.https-address" : "c6401.ambari.apache.org:50470",
          "dfs.namenode.rpc-address" : "c6401.ambari.apache.org:8020",
          "dfs.namenode.secondary.http-address" : "c6402.ambari.apache.org:50090",
          "dfs.webhdfs.enabled" : "true"
        },
        "properties_attributes" : {
          "final" : {
            "dfs.webhdfs.enabled" : "true",
            "dfs.namenode.http-address" : "true",
            "dfs.support.append" : "true",
            "dfs.namenode.name.dir" : "true",
            "dfs.datanode.failed.volumes.tolerated" : "true",
            "dfs.datanode.data.dir" : "true"
          }
        }
      }
        ]
      },
      {
    "href" : "http://AMBARI_ADDRESS/api/v1/clusters/CLUSTER_NAME/configurations/service_config_versions?
service_name=YARN&service_config_version=1",
    "cluster_name" : "CLUSTER_NAME",
    "configurations" : [
      {
        "Config" : {
          "cluster_name" : "CLUSTER_NAME",
          "stack_id" : "HDP-2.6"
        },
        "type" : "yarn-site",
        "properties" : {
          "yarn.http.policy" : "HTTP_ONLY",
          "yarn.log.server.url" : "http://c6402.ambari.apache.org:19888/jobhistory/logs",
          "yarn.log.server.web-service.url" : "http://c6402.ambari.apache.org:8188/ws/v1/applicationhistory",
          "yarn.nodemanager.address" : "0.0.0.0:45454",
          "yarn.resourcemanager.address" : "c6402.ambari.apache.org:8050",
          "yarn.resourcemanager.admin.address" : "c6402.ambari.apache.org:8141",
          "yarn.resourcemanager.ha.enabled" : "false",
          "yarn.resourcemanager.hostname" : "c6402.ambari.apache.org",
          "yarn.resourcemanager.webapp.address" : "c6402.ambari.apache.org:8088",
          "yarn.resourcemanager.webapp.delegation-token-auth-filter.enabled" : "false",
          "yarn.resourcemanager.webapp.https.address" : "c6402.ambari.apache.org:8090",
                },
        "properties_attributes" : { }
```

```
        },
          ]
        },
]
```

Cluster Service Components - /api/v1/clusters/CLUSTER_NAME/components

/api/v1/clusters/CLUSTER_NAME/components

```
"items" : [
  {
    "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/CLUSTER_NAME/components/HCAT",
    "ServiceComponentInfo" : {
      "cluster_name" : "CLUSTER_NAME",
      "component_name" : "HCAT",
      "service_name" : "HIVE"
    }
  },
  {
    "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/CLUSTER_NAME/components/HIVE_SERVER",
    "ServiceComponentInfo" : {
      "cluster_name" : "CLUSTER_NAME",
      "component_name" : "HIVE_SERVER",
      "service_name" : "HIVE"
    }
  },
  {
    "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/CLUSTER_NAME/components/NAMENODE",
    "ServiceComponentInfo" : {
      "cluster_name" : "CLUSTER_NAME",
      "component_name" : "NAMENODE",
      "service_name" : "HDFS"
    }
  },
  {
    "href" : "http://c6401.ambari.apache.org:8080/api/v1/clusters/CLUSTER_NAME/components/OOZIE_SERVER",
    "ServiceComponentInfo" : {
      "cluster_name" : "CLUSTER_NAME",
      "component_name" : "OOZIE_SERVER",
      "service_name" : "OOZIE"
    }
  }
]
```

4.2 ZooKeeper Service Registry

- TODO: provide some sense of completeness, APIs and examples

4.3 Topology Change Discovery

Since the service URLs for a cluster will be discovered, Knox has the opportunity to respond dynamically to subsequent topology changes. For a Knox topology that has been generated and deployed, it's possible that the URL for a given service could change at some point afterward.
The host name could change. The scheme and/or port could change (e.g., http --> https). The potential and frequency of such changes certainly varies among deployments.
We should consider providing the *option* for Knox to detect topology changes for a cluster, and respond by updating its corresponding topology.

For example, Ambari provides the ability to request the active configuration versions for all the service components in a cluster. There could be a thread that checks this set, notices one or more version changes, and initiates the re-generation/deployment of that topology.

Another associated benefit is the capability for Knox to interoperate with Ambari instances that are unaware of the Knox instance. Knox no longer MUST be managed by Ambari.

5. Provider Configurations

KIP-1 touched on an improvement for Centralized Gateway Configuration for LDAP/AD and the ability to import topology fragments into other topologies by referencing them by name within the topology.

While we haven't actually implemented anything along these lines yet, we are touching on something very similar in this proposal. We need to be able to author a set of Provider Configurations that can be referenced by name in the simple descriptor, that can be enumerated for inclusion in a dropdown for UIs that need to be able to select a Provider Configuration and a set of APIs added to the Admin API for management of the configuration with CRUD operations.

Whether we import the common Provider Configuration or pull in the contents of it into a generated topology can certainly be a design discussion. I think that importing at runtime would be best as we wouldn't have to regenerate all the topologies that included them if something is changed in the Provider Configuration. Though there would need to be some way to reimport the Provider Config at runtime if something changes.

6. Discovery Service Authentication

The discovery service must be authenticated by at least some of the service registries (e.g., Ambari). We need to define the means by which credentials are configured for this service in Knox.

6.1 Ambari

HTTP Basic authentication is supported by Ambari, so the Ambari service discovery implementation can leverage the Knox Alias service to obtain the necessary credentials at discovery time.

The username can be specified in a descriptor, using the discovery-user property. The default user name can also be mapped to the alias ambari.discovery.user.

Similarly, the password alias can be specified in a descriptor, using the discovery-pwd-alias property. By default, the Ambari service discovery will lookup the ambari.discovery.password alias to get the password.

So, there are two ways to specify the username for Ambari interaction:

1. Provision the alias mapping using the knoxcli.sh script

   bin/knoxcli.sh create-alias ambari.discovery.user --value ambariuser

2. Specify the discovery-user property in a descriptor (This can be useful if a Knox instance will proxy services in clusters managed by multiple Ambari instances)

   "discovery-user":"ambariuser"

And, two ways to specify the associated password:

1. Provision the password mapped to the default alias, ***ambari.discovery.password***

   bin/knoxcli.sh create-alias ambari.discovery.password --value ambaripasswd

2. Provision a different alias, and specify it in the descriptor (This can be useful if a Knox instance will proxy services in clusters managed by multiple Ambari instances)

   "discovery-pwd-alias":"my.ambari.discovery.password.alias"

Related Links

- Topology Policy Separation