Apache Ambari 2.4 – Stack/Extension/Service Definitions

Table of Contents

Introduction

Ambari supports the concept of Stacks and associated Services in a Stack Definition. By leveraging the Stack Definition, Ambari has a consistent and defined interface to install, manage and monitor a set of Services and provides an extensibility model for new Stacks and Services to be introduced. There is also support for the concept of Extensions and its associated custom Services in an Extension Definition.

Terminology

Stack

Defines a set of Services and where to obtain the software packages for those Services. A Stack can have one or more versions, and each version can be active/inactive. For example, Stack = "HDP-2.4".

Extension

Defines a set of custom Services which can be added to a stack version. An Extension can have one or more versions.

Service

Defines the Components (MASTER, SLAVE, CLIENT) that make up the Service. For example, Service = "HDFS".

Component

The individual Components that adhere to a certain defined lifecycle (start, stop, install, etc). For example, Service = "HDFS" has Components = "NameNode (MASTER)", "Secondary NameNode (MASTER)", "DataNode (SLAVE)" and "HDFS Client (CLIENT)".

Stacks

Stack Versions

Each stack-version (Example: HDP-2.3, HDP-2.4) must provide a metainfo.xml descriptor file which describes the following about this stack-version:

```
<metainfo>
    <versions>
      <active>true</active>
    </versions>
    <extends>2.3</extends>
    <minJdk>1.7</minJdk>
    <maxJdk>1.8</maxJdk>
</metainfo>
```

- **versions/active** - Whether this stack-version is still available for install. If not available, this version will not show up in UI during install.
- **extends** - The stack-version in this stack that is being extended. Extended stack-versions inherit services along with almost all aspects of the parent stack-version.
- **minJdk** - Minimum JDK with which this stack-version is supported. Users are warned during installer wizard if the JDK used by Ambari is lower than this version.
- **maxJdk** - Maximum JDK with which this stack-version is supported. Users are warned during installer wizard if the JDK used by Ambari is greater than this version.

Stack Structure

The structure of a Stack definition is as follows:

```
|_ stacks
   |_ <stack_name>
      |_ <stack_version>
         |_ metainfo.xml
         |_ configuration
            |_ cluster-env.xml
         |_ hooks
         |_ properties
            |_ stack_features.json
            |_ stack_tools.json
         |_ repos
            |_ repoinfo.xml
         |_ services
            |_ <service_name>
               |_ {files and directories}
```

Stack Properties

Similar to stack configurations, most properties are defined at the service level, however there are global properties which can be defined at the stack-version level affecting across all services.

Some examples are: stack-selector and conf-selector specific names or what stack versions certain stack features are supported by. Such properties can be defined in JSON format in the properties folder of the stack.

These properties must be defined in the base stack version. Stack versions which inherit from the base stack version MUST NOT override these settings.

Stack Features

Stacks can support different features depending on their version, for example: upgrade support, NFS support, support for specific new components such as Ranger or Phoenix.

The stack features are included in the configuration/cluster-env.xml file at the root stack version level.

```
<property>
  <name>stack_features</name>
  <value/>
  <description>List of features supported by the stack</description>
  <property-type>VALUE_FROM_PROPERTY_FILE</property-type>
  <value-attributes>
    <property-file-name>stack_features.json</property-file-name>
    <property-file-type>json</property-file-type>
    <read-only>true</read-only>
    <overridable>false</overridable>
    <visible>false</visible>
  </value-attributes>
  <on-ambari-upgrade add="true"/>
</property>
```

Stack features properties should be defined in the properties/stack_features.json file at the root stack version level. The following is an example of features described in the stack_features.json file:

```
{
  "stack_features": [
    {
      "name": "snappy",
      "description": "Snappy compressor/decompressor support",
      "min_version": "2.0.0.0",
      "max_version": "2.2.0.0"
    },
    {
      "name": "express_upgrade",
      "description": "Express upgrade support",
      "min_version": "2.1.0.0"
    }
  ]
}
```

where min_version/max_version are optional constraints.

Stack feature constants, matching features names, such has ROLLING_UPGRADE = "rolling_upgrade" have been added to a new StackFeature class in the constants.py file.

```
class StackFeature:
  """
  Stack Feature supported
  """
  SNAPPY = "snappy"
  LZO = "lzo"
  EXPRESS_UPGRADE = "express_upgrade"
  ROLLING_UPGRADE = "rolling_upgrade"
```

Additionally, corresponding helper functions has been introduced in stack_features.py to parse the JSON file content and can be called from service scripts to check if the stack supports specific features.

Example service script:

```
if params.version and check_stack_feature(StackFeature.ROLLING_UPGRADE,
params.version):

  conf_select.select(params.stack_name, "hive", params.version)

  stack_select.select("hive-server2", params.version)
```

Stack Tools

Similar to stack features, stack-selector and conf-selector tools are now stack-driven instead of hardcoding hdp-select and conf-select. They are defined in stack_tools.json file. Similarly with stack features they are declared in the configuration/cluster-env.xml file at the root stack version level.

```
<property>
  <name>stack_tools</name>
  <value/>
  <description>Stack specific tools</description>
  <property-type>VALUE_FROM_PROPERTY_FILE</property-type>
  <value-attributes>
    <property-file-name>stack_tools.json</property-file-name>
    <property-file-type>json</property-file-type>
    <read-only>true</read-only>
    <overridable>false</overridable>
    <visible>false</visible>
  </value-attributes>
  <on-ambari-upgrade add="true"/>
</property>
```

Corresponding helper functions have been added in the stack_tools.py file.

Stack Inheritance

Stacks can extend other Stacks in order to share command scripts and configurations. This reduces duplication of code across Stacks with the following:

- define repositories for the child Stack
- add new Services in the child Stack (not in the parent Stack)
- override command scripts of the parent Services
- override configurations of the parent Services

When a stack inherits from another stack version, how its defining files and directories are inherited follows a number of different patterns.

As previously mentioned the following files should not be redefined at the child stack version level:

```
properties/stack_features.json
properties/stack_tools.json
```

Note: These files should only exist at the base stack level.

The following files if defined in the current stack version replace the definitions from the parent stack version:

```
kerberos.json
widgets.json
```

The following files if defined in the current service version are merged with the parent service version:

```
configuration/cluster-env.xml
role_command_order.json
```

Note: All the services' role command orders will be merge with the stack's role command order to provide a master list.

All attributes of the current stack version's metainfo.xml will replace those defined in the parent stack version.

The following directories if defined in the current service version replace those from the parent service version:

```
hooks
```

This means the files included in those directories at the parent level will not be inherited. You will need to copy all the files you wish to keep from that directory structure.

The following directories are not inherited:

```
repos
upgrades
```

The repos/repoinfo.xml file must be defined in every active stack version. The upgrades directory and its corresponding XML files should be defined in all stack versions that support upgrade.

Services Folder Inheritance

The services folder is a special case. There are two inheritance mechanisms at work here. First the stack_advisor.py will automatically import the parent stack version's stack_advisor.py script but the rest of the inheritance behavior is up to the script's author. There are several examples of stack_advisor.py files in the Ambari server source.

```
class HDP23StackAdvisor(HDP22StackAdvisor):
  def __init__(self):
    super(HDP23StackAdvisor, self).__init__()
    Logger.initialize_logger()

  def getComponentLayoutValidations(self, services, hosts):
    parentItems = super(HDP23StackAdvisor, self).getComponentLayoutValidations(services, hosts)
    ...
```

Services defined within the services folder follow the rules for service inheritance. By default, if a service does not declare an explicit inheritance (via the extends tag), the service will inherit from the service defined at the parent stack version.

Extensions

An Extension is a collection of one or more custom services which are packaged together. Much like stacks, each extension has a name which needs to be unique in the cluster. It also has a version directory to distinguish different releases of the extension. Much like stack versions which go in /var/lib/ambari-server/resources/stacks with <stack_name>/<stack_version> sub-directories, extension versions go in /var/lib/ambari-server/resources/extensions with <extension_name>/<extension_version> sub-directories.

An extension can be linked to supported stack versions. Once an extension version has been linked to the currently installed stack version, the custom services contained in the extension version may be added to the cluster in the same manner as if they were actually contained in the stack version.

Third party developers can release Extensions which can be added to a cluster.

Extension Structure

The structure of an Extension definition is as follows:

```
|_ extensions
   |_ <extension_name>
      |_ <extension_version>
         |_ metainfo.xml
         |_ services
            |_ <service_name>
               |_ metainfo.xml
               |_ metrics.json
               |_ configuration
                  |_ {configuration files}
               |_ package
                  |_ {files, scripts, templates}
```

An extension version is similar to a stack version but it only includes the metainfo.xml and the services directory. This means that the alerts, kerberos, metrics, role command order, widgets files are not supported and should be included at the service level. In addition, the repositories, hooks, configurations, and upgrades directories are not supported although upgrade support can be added at the service level.

Extension Inheritance

Extension versions can extend other Extension versions in order to share command scripts and configurations. This reduces duplication of code across Extensions with the following:

- add new Services in the child Extension version (not in the parent Extension version)
- override command scripts of the parent Services
- override configurations of the parent Services

For example, MyExtension 2.0 could extend MyExtension 1.0 so only the changes applicable to the MyExtension 2.0 extension are present in that Extension definition. This extension is defined in the metainfo.xml for MyExtension 2.0:

```
<metainfo>
  <extends>1.0</extends>
```

Supported Stack Versions

Each Extension Version must support one or more Stack Versions. The Extension Version specifies the minimum Stack Version which it supports. This is included in the extension's metainfo.xml in the prerequisites section like so:

```
<metainfo>
  <prerequisites>
    <min-stack-versions>
      <stack>
        <name>HDP</name>
        <version>2.4</version>
      </stack>
      <stack>
        <name>OTHER</name>
        <version>1.0</version>
      </stack>
    </min-stack-versions>
  </prerequisites>
</metainfo>
```

Extension Links

An Extension Link is a link between a stack version and an extension version. Once an extension version has been linked to the currently installed stack version, the custom services contained in the extension version may be added to the cluster in the same manner as if they were actually contained in the stack version.

It is only possible to link an extension version to a stack version if the stack version is supported by the extension version. The stack name must be specified in the prerequisites section of the extension's metainfo.xml and the stack version must be greater than or equal to the minimum version number specified.

Services

Service Overview

The metainfo.xml file in a Service describes the service, the components of the service and the management scripts to use for executing commands.

Metainfo

The metainfo.xml file is a declarative definition of an Ambari managed service describing its content. It is the most critical file of a service definition.

Structure

Note: In this section non-mandatory fields are in italics.

The fields to describe a **service** are as follows:

- **name**: the name of the service. A name has to be unique among all the services that are included in the stack definition containing the service.
- **displayName**: the display name of the service
- **version**: the version of the service. name and version together uniquely identify a service. Usually, the version is the version of the service binary itself.
- **components**: the list of component that the service is comprised of
- **osSpecifics**: OS specific package information for the service
- *commandScript*: service level commands may also be defined. The command is executed on a component instance that is a client
- *comment*: a short description describing the service
- *requiredServices*: what other services that should be present on the cluster
- *configuration-dependencies*: configuration files that are expected by the service (config files owned by other services are specified in this list)

service/components - A service contains several components. The fields associated with a component are:

- **name**: name of the component
- **category**: type of the component - MASTER, SLAVE, and CLIENT
- **commandScript**: application wide commands may also be defined. The command is executed on a component instance that is a client
- *cardinality*: allowed/expected number of instances
- *versionAdvertised*: does the component advertise its version - used during rolling/express upgrade
- *timelineAppid*: the default category used to store generated metrics data
- *dependencies*: the list of components that this component depends on
- *customCommands*: a set of custom commands associated with the component in addition to standard commands

service/osSpecifics - OS specific package names (rpm or deb packages)

- **osFamily**: the os family for which the package is applicable
- **packages**: list of packages that are needed to deploy the service
- **package/name**: name of the package (will be used by the yum/zypper/apt commands)

service/commandScript - the script that implements service check (see service/component/customCommand below)

service/component/commandScript - the script that implements components specific default commands (see service/component/customCommand below)

service/component/customCommand - custom commands can be added to components.

- **name**: name of the custom command
- **commandScript**: the details of the script that implements the custom command
- **commandScript/script**: the relative path to the script
- *commandScript/scriptType*: the type of the script, currently only supported type is PYTHON
- *commandScript/timeout*: custom timeout for the command - this supersedes ambari default

service/component/configFiles - list of config files to be available when client config is to be downloaded (used to configure service clients that are not managed by Ambari)

- **type**: the type of file to be generated, xml or env sh, yaml, etc
- **fileName**: name of the generated file
- **dictionary**: data dictionary that contains the config properties (relevant to how ambari manages config bags internally)

Components and Commands

A component of a service must be either a MASTER, SLAVE or CLIENT category. The <category> tells Ambari what default commands should be available to manage and monitor the component. Details of various sections in metainfo.xml can be found in the Writing metainfo.xml section.

For each Component you must specify the <commandScript> to use when executing commands. There is a defined set of default commands the component must support depending on the components category.

MASTER	install, start, stop, configure, status
SLAVE	install, start, stop, configure, status
CLIENT	install, configure, status

Service Scripts

Ambari supports different commands scripts written in PYTHON. The type is used to know how to execute the command scripts. You can also create custom commands if there are other commands beyond the default lifecycle commands your component needs to support.

For example, in the YARN Service describes the ResourceManager component as follows in metainfo.xml:

```
<component>
  <name>RESOURCEMANAGER</name>
  <category>MASTER</category>
  <commandScript>
    <script>scripts/resourcemanager.py</script>
    <scriptType>PYTHON</scriptType>
    <timeout>600</timeout>
  </commandScript>
  <customCommands>
    <customCommand>
      <name>DECOMMISSION</name>
      <commandScript>
        <script>scripts/resourcemanager.py</script>
        <scriptType>PYTHON</scriptType>
        <timeout>600</timeout>
      </commandScript>
    </customCommand>
  </customCommands>
</component>
```

The ResourceManager is a MASTER component, and the command script is scripts/resourcemanager.py, which can be found in the services/YARN/package directory. That command script is PYTHON and that script implements the default lifecycle commands as python methods. This is the install method for the default INSTALL command:

```
class Resourcemanager(Script):
  def install(self, env):
    self.install_packages(env)
    self.configure(env)
```

You can also see a custom command is defined DECOMMISSION, which means there is also a decommission method in that python command script:

```
def decommission(self, env):
  import params
  ...
  Execute(yarn_refresh_cmd,
          user=yarn_user
  )
  Pass
```

Creating Custom Services

In this example, we will create a custom service called "SAMPLESRV". This service includes MASTER, SLAVE and CLIENT components.

First, create a directory named SAMPLESRV that will contain the service definition for SAMPLESRV.

```
mkdir SAMPLESRV
cd SAMPLESRV
```

Within the SAMPLESRV directory, create a metainfo.xml file that describes the new service. For example:

```
<?xml version="1.0"?>
<metainfo>
    <schemaVersion>2.0</schemaVersion>
    <services>
        <service>
            <name>SAMPLESRV</name>
            <displayName>New Sample Service</displayName>
            <comment>A New Sample Service</comment>
            <version>1.0.0</version>
            <components>
                <component>
                    <name>SAMPLESRV_MASTER</name>
                    <displayName>Sample Srv Master</displayName>
                    <category>MASTER</category>
                    <cardinality>1</cardinality>
                    <commandScript>
                        <script>scripts/master.py</script>
                        <scriptType>PYTHON</scriptType>
                        <timeout>600</timeout>
                    </commandScript>
                </component>
                <component>
                    <name>SAMPLESRV_SLAVE</name>
                    <displayName>Sample Srv Slave</displayName>
                    <category>SLAVE</category>
                    <cardinality>1+</cardinality>
                    <commandScript>
                        <script>scripts/slave.py</script>
                        <scriptType>PYTHON</scriptType>
                        <timeout>600</timeout>
                    </commandScript>
                </component>
                <component>
                    <name>SAMPLESRV_CLIENT</name>
                    <displayName>Sample Srv Client</displayName>
                    <category>CLIENT</category>
                    <cardinality>1+</cardinality>
                    <commandScript>
                        <script>scripts/sample_client.py</script>
                        <scriptType>PYTHON</scriptType>
                        <timeout>600</timeout>
                    </commandScript>
                </component>
            </components>
            <osSpecifics>
                <osSpecific>
                    <osFamily>any</osFamily>
                </osSpecific>
            </osSpecifics>
        </service>
    </services>
</metainfo>
```

In the above, the service name is "SAMPLESRV", and it contains:

one MASTER component "SAMPLESRV_MASTER"
one SLAVE component "SAMPLESRV_SLAVE"
one CLIENT component "SAMPLESRV_CLIENT"

Next, create that command script. Create a directory for the command script SAMPLESRV/package/scripts that we designated in the service metainfo.xml.

```
mkdir -p package/scripts
cd package/scripts
```

Within the scripts directory, create the python command script files mentioned in the metainfo.xml.

An example master.py file:

```
import sys
from resource_management import *
class Master(Script):
  def install(self, env):
    print 'Install the Sample Srv Master';
  def configure(self, env):
    print 'Configure the Sample Srv Master';
  def stop(self, env):
    print 'Stop the Sample Srv Master';
  def start(self, env):
    print 'Start the Sample Srv Master';
  def status(self, env):
    print 'Status of the Sample Srv Master';
if __name__ == "__main__":
  Master().execute()
```

An example slave.py file:

```
import sys
from resource_management import *
class Slave(Script):
  def install(self, env):
    print 'Install the Sample Srv Slave';
  def configure(self, env):
    print 'Configure the Sample Srv Slave';
  def stop(self, env):
    print 'Stop the Sample Srv Slave';
  def start(self, env):
    print 'Start the Sample Srv Slave';
  def status(self, env):
    print 'Status of the Sample Srv Slave';
if __name__ == "__main__":
  Slave().execute()
```

An example sample_client.py file:

```
import sys
from resource_management import *
class SampleClient(Script):
  def install(self, env):
    print 'Install the Sample Srv Client';
  def configure(self, env):
    print 'Configure the Sample Srv Client';
if __name__ == "__main__":
  SampleClient().execute()
```

Implementing Custom Commands

Browse to the SAMPLESRV directory, and edit the metainfo.xml file that describes the service. For example, adding a custom command to the SAMPLESRV_CLIENT:

```
<component>
    <name>SAMPLESRV_CLIENT</name>
    <displayName>Sample Srv Client</displayName>
    <category>CLIENT</category>
    <cardinality>1+</cardinality>
    <commandScript>
        <script>scripts/sample_client.py</script>
        <scriptType>PYTHON</scriptType>
        <timeout>600</timeout>
    </commandScript>
    <customCommands>
      <customCommand>
        <name>SOMETHINGCUSTOM</name>
        <commandScript>
          <script>scripts/sample_client.py</script>
          <scriptType>PYTHON</scriptType>
          <timeout>600</timeout>
        </commandScript>
      </customCommand>
    </customCommands>
</component>
```

Next, create that command script by editing the package/scripts/sample_client.py file that we designated in the service metainfo.xml.

```
import sys
from resource_management import *
class SampleClient(Script):
  def install(self, env):
    print 'Install the Sample Srv Client';
  def configure(self, env):
    print 'Configure the Sample Srv Client';
  def somethingcustom(self, env):
    print 'Something custom';

if __name__ == "__main__":
  SampleClient().execute()
```

Adding Configuration Settings to a Custom Service

In this example, we will add a configuration type "test-config" to our SAMPLESRV. First, modify the metainfo.xml

```
<component>
    <name>SAMPLESRV_CLIENT</name>
    <displayName>Sample Srv Client</displayName>
    <category>CLIENT</category>
    <cardinality>1+</cardinality>
    <commandScript>
        <script>scripts/sample_client.py</script>
        <scriptType>PYTHON</scriptType>
        <timeout>600</timeout>
    </commandScript>
    <configFiles>
      <configFile>
        <type>xml</type>
        <fileName>test-config.xml</fileName>
        <dictionaryName>test-config</dictionaryName>
      </configFile>
    </configFiles>
</component>
```

Create a directory for the configuration dictionary file SAMPLESRV/configuration.

```
mkdir -p configuration
cd configuration
```

Create the test-config.xml file. For example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>some.test.property</name>
    <value>this.is.the.default.value</value>
    <description>This is a test description.</description>
  </property>
  <property>
    <name>another.test.property</name>
    <value>5</value>
    <description>This is a second test description.</description>
  </property>
</configuration>
```

There is an optional setting "configuration-dir". Custom services should either not include the setting or should leave it as the default value "configuration".

```
<configuration-dir>configuration</configuration-dir>
```

Configuration dependencies can be included in the metainfo.xml in the a "configuration-dependencies" section. This section can be added to the service as a whole or a particular component. One of the implications of this dependency is that whenever the config-type is updated, Ambari automatically marks the component or service as requiring restart.

```
<configuration-dependencies>
  <config-type>core-site</config-type>
  <config-type>hdfs-site</config-type>
</configuration-dependencies>
```

Service Advisor

Each custom service can provide a service advisor as a Python script named service-advisor.py in their service folder. A Service Advisor allows custom services to integrate into the stack advisor behavior.

Service Advisor Inheritance

Unlike the Stack-advisor scripts, the service-advisor scripts do not automatically extend the parent service's service-advisor scripts. The service-advisor script needs to explicitly extend their parent's service service-advisor script. The following code sample shows how you would refer to a parent's service_advisor.py. In this case it is extending the root service-advisor.py file in the resources/stacks directory.

```
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
STACKS_DIR = os.path.join(SCRIPT_DIR, '../../../stacks/')
PARENT_FILE = os.path.join(STACKS_DIR, 'service_advisor.py')

try:
  with open(PARENT_FILE, 'rb') as fp:
    service_advisor = imp.load_module('service_advisor', fp, PARENT_FILE,
('.py', 'rb', imp.PY_SOURCE))
except Exception as e:
  traceback.print_exc()
  print "Failed to load parent"

class HAWQ200ServiceAdvisor(service_advisor.ServiceAdvisor):
```

Service Advisor Behavior

Like the stack advisors, service advisors provide information on 4 important aspects for the service:

- Recommend layout of the service on cluster
- Recommend service configurations
- Validate layout of the service on cluster
- Validate service configurations

By providing the service-advisor.py file, one can control dynamically each of the above for the service.

The main interface for the service-advisor scripts contains documentation on how each of the above are called, and what data is provided.

```
class ServiceAdvisor(DefaultStackAdvisor):

  def colocateService(self, hostsComponentsMap, serviceComponents):
    pass

  def getServiceConfigurationRecommendations(self, configurations,
clusterSummary, services, hosts):
    pass

  def getServiceComponentLayoutValidations(self, services, hosts):
    return []

  def getServiceConfigurationsValidationItems(self, configurations,
recommendedDefaults, services, hosts):
    return []
```

Service Advisor Examples

Service Advisor interface

HAWQ 2.0.0 Service Advisor implementation

PXF 3.0.0 Service Advisor implementation

Service Upgrade

Each custom service can define its upgrade within its service definition. This allows the custom service to be integrated within the stack's upgrade.

Service Upgrade Packs

Each service can define upgrade-packs, which are XML files describing the upgrade process of that particular service and how the upgrade pack relates to the overall stack upgrade-packs. These upgrade-pack XML files are placed in the service's upgrades/ folder in separate sub-folders specific to the stack-version they are meant to extend. Some examples of this can be seen in the testing code.

Upgrades folder

Upgrade-pack XML

Matching Upgrade Packs

Each upgrade-pack that the service defines should match the file name of the service defined by a particular stack version. For example in the testing code, HDP 2.2.0 had an upgrade_test_15388.xml upgrade-pack. The HDFS service defined an extension to that upgrade pack HDP/2.0.5/services/HDFS/upgrades/HDP/2.2.0/upgrade_test_15388.xml. In this case the upgrade-pack was defined in the HDP/2.0.5 stack. The upgrade-pack is an extension to HDP/2.2.0 because it is defined in upgrade/HDP/2.2.0 directory. Finally, the name of the service's extension to the upgrade-pack upgrade_test_15388.xml matches the name of the upgrade-pack in HDP/2.2.0/upgrades.

Upgrade XML Format

The file format for the service is much the same as that of the stack. The target, target-stack and type attributes should all be the same as the stack's upgrade-pack.

Prerequisite Checks

The service is able to add its own prerequisite checks.

```
<upgrade xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <target>2.4.*</target>
  <target-stack>HDP-2.4.0</target-stack>
  <type>ROLLING</type>
  <prerequisite-checks>
    <check>org.apache.ambari.server.checks.FooCheck</check>
  </prerequisite-checks>
```

Order Section

The order section of the upgrade-pack, consists of group elements just like the stack's upgrade-pack. The key difference is defining how these groups relate to groups in the stack's upgrade pack or other service upgrade-packs. In the first example we are referencing the PRE_CLUSTER group and adding a new execute-stage for the service FOO. The entry is supposed to be added after the execute-stage for HDFS based on the <add-after-group-entry> tag.

```
<order>
  <group xsi:type="cluster" name="PRE_CLUSTER" title="Pre
{{direction.text.proper}}">
    <add-after-group-entry>HDFS</add-after-group-entry>
    <execute-stage service="FOO" component="BAR" title="Backup FOO">
      <task xsi:type="manual">
        <message>Back FOO up.</message>
      </task>
    </execute-stage>
  </group>
```

The same syntax can be used to order other sections like service check priorities and group services.

```
<group name="SERVICE_CHECK1" title="All Service Checks" xsi:type="service-
check">
  <add-after-group-entry>ZOOKEEPER</add-after-group-entry>
  <priority>
    <service>HBASE</service>
  </priority>
</group>

<group name="CORE_MASTER" title="Core Masters">
  <add-after-group-entry>YARN</add-after-group-entry>
  <service name="HBASE">
    <component>HBASE_MASTER</component>
  </service>
</group>
```

It is also possible to add new groups and order them after other groups in the stack's upgrade-packs. In the following example, we are adding the FOO group after the HIVE group using the add-after-group tag.

```
<group name="FOO" title="Foo">
  <add-after-group>HIVE</add-after-group>
  <skippable>true</skippable>
  <allow-retry>false</allow-retry>
  <service name="FOO">
    <component>BAR</component>
  </service>
</group>
```

You could also include both the add-after-group and the add-after-group-entry tags in the same group. This will create a new group if it doesn't already exist and will order it after the add-after-group's group name. The add-after-group-entry will determine the internal ordering of that group's services, priorities or execute stages.

```
<group name="FOO" title="Foo">
  <add-after-group>HIVE</add-after-group>
  <add-after-group-entry>FOO</add-after-group-entry>
```

```
  <skippable>true</skippable>
  <allow-retry>false</allow-retry>
  <service name="FOO2">
    <component>BAR2</component>
  </service>
</group>
```

Processing Section

The processing section of the upgrade-pack remains the same as what it would be in the stack's upgrade-pack.

```
<processing>
  <service name="FOO">
    <component name="BAR">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
    <component name="BAR2">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
  </service>
</processing>
```

Service Role Command Order

Each service can define its own role command order by including a role_command_order.json file in its service folder. The service should only specify the relationship of its components to other components. In other words, if a service only includes COMP_X, it should only list dependencies related to COMP_X. If when COMP_X starts it is dependent on the NameNode start and when the NameNode stops it should wait for COMP_X to stop, the following would be included in the role command order:

```
{
  "_comment" : "Record format:",
  "_comment" : "blockedRole-blockedCommand: [blockerRole1-blockerCommand1,
blockerRole2-blockerCommand2, ...]",
  "general_deps" : {
    "_comment" : "dependencies for all cases"
  },
  "_comment" : "Dependencies that are used when GLUSTERFS is not present in
cluster",
  "optional_no_glusterfs": {
    "COMP_X-START": ["NAMENODE-START"],
    "NAMENODE-STOP": ["COMP_X-STOP"]
  }
}
```

The entries in the service's role command order will be merged with the role command order defined in the stack. For example, since the stack already has a dependency for NAMENODE-STOP, in the example above COMP_X-STOP would be added to the rest of the NAMENODE-STOP dependencies and the COMP_X-START dependency on NAMENODE-START would be added as a new dependency.

Role Command Order Sections

Ambari uses the below sections only:

general_deps	Command orders are applied in all situations
optional_glusterfs	Command orders are applied when cluster has instance of GLUSTERFS service
optional_no_glusterfs	Command orders are applied when cluster does not have instance of GLUSTERFS service
namenode_optional_ha	Command orders are applied when HDFS service is installed and JOURNALNODE component exists (HDFS HA is enabled)
resourcemanager_optional_ha	Command orders are applied when YARN service is installed and multiple RESOURCEMANAGER host-components exist (YARN HA is enabled)

Commands

Commands currently supported by Ambari are:

```
INSTALL
UNINSTALL
START
RESTART
STOP
EXECUTE
ABORT
UPGRADE
SERVICE_CHECK
CUSTOM_COMMAND
ACTIONEXECUTE
```

Alerts

Each service is capable of defining which alerts Ambari should track by providing an alerts.json file.

Alert definitions are the templates that are used to distribute alerts to the appropriate Ambari agents. They govern the type of alert, the threshold values, and the information to be used when notifying a target. A single definition can be distributed to more than one host in order to produce multiple instances of an alert.

Each definition contains common information, such as a unique identifier, service, and component. Beyond this, definitions declare a type with each type having its own distinct properties.

Common Properties

- id
- name
- label
- cluster_name
- service_name
- component_name
- source

Types

Port

Port definitions are used to check TCP connectivity to a remote endpoint.

```
"source" : {

   "default_port" : 2181,
   "reporting" : {
     "ok" : {
       "text" : "TCP OK - {0:.3f}s response on port {1}"
     },
     "warning" : {
       "text" : "TCP OK - {0:.3f}s response on port {1}",
       "value" : 1.5
     },
     "critical" : {
       "text" : "Connection failed: {0} to {1}:{2}",
       "value" : 5.0
     }
   },
   "type" : "PORT",
   "uri" : "{{core-site/ha.zookeeper.quorum}}"
 }
}
```

Script

Script definitions defer all functionality to a Python script accessible to the Ambari agents from a specified relative or absolute path.

```
          "source" : {
            "path" : "HDFS/2.1.0.2.0/package/alerts/alert_ha_namenode_health.py",
            "type" : "SCRIPT"
          }
```

Metric

Metric source fields are used to define JMX endpoints that can be queried for values. The source/reporting and jmx/value fields are parameterized to match the property_list specified.

```
          "source" : {
            "jmx" : {
              "property_list" : [
                "java.lang:type=OperatingSystem/SystemCpuLoad",
                "java.lang:type=OperatingSystem/AvailableProcessors"
              ],
              "value" : "{0} * 100"
            },
            "reporting" : {
              "ok" : {
                "text" : "{1} CPU, load {0:.1%}"
              },
              "warning" : {
                "text" : "{1} CPU, load {0:.1%}",
                "value" : 200.0
              },
              "critical" : {
                "text" : "{1} CPU, load {0:.1%}",
                "value" : 250.0
              },
              "units" : "%"
            },
            "type" : "METRIC",
            "uri" : {
              "http" : "{{hdfs-site/dfs.namenode.http-address}}",
              "https" : "{{hdfs-site/dfs.namenode.https-address}}",
              "https_property" : "{{hdfs-site/dfs.http.policy}}",
              "https_property_value" : "HTTPS_ONLY",
              "default_port" : 0.0,
              "high_availability" : {
                "nameservice" : "{{hdfs-site/dfs.nameservices}}",
                "alias_key" : "{{hdfs-site/dfs.ha.namenodes.{{ha-nameservice}}}}",
                "http_pattern" : "{{hdfs-site/dfs.namenode.http-address.{{ha-
nameservice}}.{{alias}}}}",
                "https_pattern" : "{{hdfs-site/dfs.namenode.https-address.{{ha-
nameservice}}.{{alias}}}}"
              }
            }
          }
```

Web

Web definitions are similar in function to Port definitions. However, instead of checking for TCP connectivity, they also verify that a proper HTTP response code was returned.

```
      "source" : {
        "reporting" : {
          "ok" : {
            "text" : "HTTP {0} response in {2:.3f} seconds"
          },
          "warning" : {
            "text" : "HTTP {0} response in {2:.3f} seconds"
          },
          "critical" : {
            "text" : "Connection failed to {1}: {3}"
          }
        },
        "type" : "WEB",
        "uri" : {
          "http" : "{{hdfs-site/dfs.namenode.http-address}}",
          "https" : "{{hdfs-site/dfs.namenode.https-address}}",
          "https_property" : "{{hdfs-site/dfs.http.policy}}",
          "https_property_value" : "HTTPS_ONLY",
          "kerberos_keytab" : "{{hdfs-
site/dfs.web.authentication.kerberos.keytab}}",
          "kerberos_principal" : "{{hdfs-
site/dfs.web.authentication.kerberos.principal}}",
          "default_port" : 0.0,
          "high_availability" : {
            "nameservice" : "{{hdfs-site/dfs.nameservices}}",
            "alias_key" : "{{hdfs-site/dfs.ha.namenodes.{{ha-nameservice}}}}",
            "http_pattern" : "{{hdfs-site/dfs.namenode.http-address.{{ha-
nameservice}}.{{alias}}}}",
            "https_pattern" : "{{hdfs-site/dfs.namenode.https-address.{{ha-
nameservice}}.{{alias}}}}"
          }
        }
      }
```

Aggregate

Aggregate definitions are used to combine the results of another alert definition from different nodes. The source/alert_name field must match the name field of another alert definition.

```
      "source": {
        "type": "AGGREGATE",
        "alert_name": "datanode_process",
        "reporting": {
          "ok": {
            "text": "affected: [{1}], total: [{0}]"
          },
          "warning": {
            "text": "affected: [{1}], total: [{0}]",
            "value": 10
          },
          "critical": {
            "text": "affected: [{1}], total: [{0}]",
            "value": 30
          },
          "units": "%",
          "type": "PERCENT"
        }
```

Structures & Concepts

- uri - Definition types that contain a URI can depend on any number of valid subproperties. In some cases, the URI may be very simple and only include a single port. In other scenarios, the URI may be more complex and include properties for plaintext, SSL, and secure endpoints protected by Kerberos.

    - http - a property that contains the plaintext endpoint to test
    - https - a property that contains the SSL endpoint to test
    - https_property - a property that contains the value which can be used to determine if the component is SSL protected
    - http_property_value - a constant value to compare with https_property in order to determine if the component is protected by SSL
    - kerberos_keytab - a property that contains the Kerberos keytab if security is enabled
    - kerberos_principal - a property that contains the Kerberos principal if security is enabled
    - default_port - a default port which can be used if none of the above properties can be realized
    - high_availability - a structure that contains a way to dynamically build properties which contain the endpoints to use when the components are running in HA mode

- reporting - a structure that defines the thresholds and text to use when determining the state for an alert. ok is always a required element, however only a single warning or critical element is needed. Some alerts may only have two states (such as OK and CRITICAL) and will bypass the need for a warning element.

- default_port - a URI, host, or integer that represents a fallback value to use if none of the other specified properties can be realized.

For an example, check out the HDFS alerts.json file.

Kerberos

Ambari is capable of enabling and disabling Kerberos for a cluster. To inform Ambari of the identities and configurations to be used for the service and its components, each service can provide a kerberos.json file.

The Kerberos Descriptor

The Kerberos Descriptor is a JSON-formatted text file containing information needed by Ambari to enable or disable Kerberos for a stack and its services. This file must be named kerberos.json and should be in the root directory of the relevant stack or service definition. Kerberos Descriptors are meant to be hierarchical such that details in the stack-level descriptor can be overwritten (or updated) by details in the service-level descriptors.

For the services in a stack to be Kerberized, there must be a stack-level Kerberos Descriptor. This ensures that even if a common service has a Kerberos Descriptor, it may not be Kerberized unless the relevant stack indicates that supports Kerberos by having a stack-level Kerberos Descriptor.

For a component of a service to be Kerberized, there must be an entry for it in its containing service's service-level descriptor. This allows for some of a services' components to be managed and other components of that service to be ignored by the automated Kerberos facility.

Kerberos Descriptors are inherited from the base stack or service, but may be overridden as a full descriptor - partial descriptors are not allowed.

A complete descriptor (which is built using the stack-level descriptor, the service-level descriptors, and any updates from user input) has the following structure:

```
Stack-level Properties
Stack-level Identities
Stack-level Configurations
Stack-level Auth-to-local-properties
Services
    Service-level Identities
    Service-level Auth-to-local-properties
    Service-level Configurations
    Components
        Component-level Identities
        Component-level Auth-to-local-properties
        Component-level Configurations
```

Each level of the descriptor inherits the data from its parent. This data, however, may be overridden if necessary. For example, a component will inherit the configurations and identities of its container service; which in turn inherits the configurations and identities from the stack.

Stack-level Properties

Stack-level properties is an optional set of name/value pairs that can be used in variable replacements. For example, if a property named "property1" exists with the value of "value1", then any instance of "${property1}" within a configuration property name or configuration property value will be replaced with "value1".

This property is only relevant in the stack-level Kerberos Descriptor and may not be overridden by lower-level descriptors. See properties for details.

Stack-level Identities

Stack-level identities is an optional identities block containing a list of zero or more identity descriptors that are common among all services in the stack. An example of such an identity is the Ambari smoke test user, which is used by all services to perform service check operations. Service- and component-level identities may reference (and specialize) stack-level identities using the identity's name with a forward slash (/) preceding it. For example if there was a stack-level identity with the name "smokeuser", then a service or a component may create an identity block that references and specializes it by declaring a "reference" property and setting it to "/smokeuser". Within this identity block details of the identity may be and overwritten as necessary. This does not alter the stack-level identity, it essentially creates a copy of it and updates the copy's properties. See identities.

Stack-level Auth-to-local-properties

Stack-level auth-to-local-properties is an optional list of zero or more configuration property specifications (config-type/property_name[|concatenation_scheme]) indicating which properties should be updated with dynamically generated auto-to-local rule sets. See auth-to-local-properties.

Stack-level Configurations

Stack-level configurations is an optional configurations block containing a list of zero or more configuration descriptors that are common among all services in the stack. Configuration descriptors are overridable due to the structure of the data. However, overriding configuration properties may create undesired behavior since it is not known until after the Kerberization process is complete what value a property will have. See configurations.

Services

Services is a list of zero or more service descriptors. A stack-level Kerberos Descriptor should not list any services; however, a service-level Kerberos Descriptor should contain at least one. See services.

Service-level Identities

Service-level identities is an optional identities block containing a list of zero or more identity descriptors that are common among all components of the service. Component-level identities may reference (and specialize) service-level identities by specifying a relative or an absolute path to it.

For example if there was a service-level identity with the name "service_identity", then a child component may create an identity block that references and specializes it by setting its "reference" attribute to "../service_identity" or "/service_name/service_identity" and overriding any values as necessary. This does not override the service-level identity, it essentially creates a copy of it and updates the copy's properties. See identities for details.

Examples:

```
{
  "name" : "relative_path_example",
  "reference" : "../service_identity",
  ...
}

{
  "name" : "absolute_path_example",
  "reference" : "/SERVICE/service_identity",
  ...
}
```

Note: By using the absolute path to an identity, any service-level identity may be referenced by any other service or component.

Service-level Auth-to-local-properties

Service-level auth-to-local-properties is an optional list of zero or more configuration property specifications (config-type/property_name[|concatenation_scheme]) indicating which properties should be updated with dynamically generated auto-to-local rule sets. See auth-to-local-properties.

Service-level Configurations

Service-level configurations is an optional configurations block listing of zero or more configuration descriptors that are common among all components within a service. Configuration descriptors may be overridden due to the structure of the data. However, overriding configuration properties may create undesired behavior since it is not known until after the Kerberization process is complete what value a property will have. See configurations.

Components

Components is a list of zero or more component descriptor blocks. See components.

Component-level Identities

Component-level identities is an optional identities block containing a list of zero or more identity descriptors that are specific to the component. A Component-level identity may be referenced (and specialized) by using the absolute path to it (/service_name/component_name/identity_name). This does not override the component-level identity, it essentially creates a copy of it and updates the copy's properties. See identities.

Component-level Auth-to-local-properties

Component-level auth-to-local-properties is an optional list of zero or more configuration property specifications (config-type/property_name[|concatenation_scheme]) indicating which properties should be updated with dynamically generated auto-to-local rule sets. See auth-to-local-properties.

Component-level Configurations

Component-level configurations is an optional configurations block listing zero or more configuration descriptors that are specific to the component. See configurations.

Descriptor Specifications

properties

The properties block is only valid in the service-level Kerberos Descriptor file. This block is a set of name/value pairs as follows:

```
"properties" : {
  "property_1" : "value_1",
  "property_2" : "value_2",
  ...
}
```

auth-to-local-properties

The auth-to-local-properties block is valid in the stack-, service-, and component-level descriptors. This block is a list of configuration specifications (config-type/property_name[|concatenation_scheme]) indicating which properties contain auth-to-local rules that should be dynamically updated based on the identities used within the Kerberized cluster.

The specification optionally declares the concatenation scheme to use to append the rules into a rule set value. If specified one of the following schemes may be set:

```
new_lines - rules in the rule set are separated by a new line (\n)
new_lines_escaped - rules in the rule set are separated by a \ and a new line
(\n)
spaces - rules in the rule set are separated by a whitespace character
(effectively placing all rules in a single line)
```

If not specified, the default concatenation scheme is new_lines.

```
"auth-to-local-properties" : [
  "core-site/hadoop.security.auth_to_local",
  "service.properties/http.authentication.kerberos.name.rules|new_lines_escaped",
  ...
]
```

configurations

A configurations block may exist in stack-, service-, and component-level descriptors. This block is a list of one or more configuration blocks containing a single structure named using the configuration type and containing values for each relevant property.

Each property name and value may be a concrete value or contain variables to be replaced using values from the stack-level properties block or any available configuration. Properties from the properties block are referenced by name (${property_name}) and configuration properties are reference by configuration specification (${config-type/property_name}).

```
"configurations" : [
  {
    "config-type-1" : {
      "${cluster-env/smokeuser}_property" : "value1",
```

```
      "some_realm_property" : "${realm}",
       ...
    }
  },
  {
    "config-type-2" : {
      "property-2" : "${cluster-env/smokeuser}",
      ...
    }
  },
  ...
]
```

If cluster-env/smokuser was "ambari-qa" and realm was "EXAMPLE.COM", the above block would effectively be translated to:

```
"configurations" : [
  {
    "config-type-1" : {
      "ambari-qa_property" : "value1",
      "some_realm_property" : "EXAMPLE.COM",
      ...
    }
  },
  {
    "config-type-2" : {
      "property-2" : "ambari-qa",
      ...
    }
  },
  ...
]
```

identities

An identities descriptor may exist in stack-, service-, and component-level descriptors. This block is a list of zero or more identity descriptors. Each identity descriptor is a block containing a name, an optional reference identifier, an optional principal descriptor, and an optional keytab descriptor.

The name property of an identity descriptor should be a concrete name that is unique with in its local scope (stack, service, or component). However, to maintain backwards-compatibility with previous versions of Ambari, it may be a reference identifier to some other identity in the Kerberos Descriptor. This feature is deprecated and may not be available in future versions of Ambari.

The reference property of an identity descriptor is optional. If it exists, it indicates that the properties from referenced identity is to be used as the base for the current identity and any properties specified in the local identity block overrides the base data. In this scenario, the base data is copied to the local identities and therefore changes are realized locally, not globally. Referenced identities may be hierarchical, so a referenced identity may reference another identity, and so on. Because of this, care must be taken not to create cyclic references. Reference values must be in the form of a relative or absolute path to the referenced identity descriptor. Relative paths start with a ../ and may be specified in component-level identity descriptors to reference an identity descriptor in the parent service.

Absolute paths start with a / and may be specified at any level as follows:

```
Stack-level identity reference: /identitiy_name
Service-level identity reference: /SERVICE_NAME/identitiy_name
Component-level identity reference: /SERVICE_NAME/COMPONENT_NAME/identitiy_name
```

```
"identities" : [
  {
    "name" : "local_identity",
    "principal" : {
      ...
    },
    "keytab" : {
      ...
    }
  },
  {
    "name" : "/smokeuser",
    "principal" : {
      "configuration" : "service-site/principal_property_name"
    },
    "keytab" : {
      "configuration" : "service-site/keytab_property_name"
    }
  },
  ...
]
```

principal

The principal block is an optional block inside an identity descriptor block. It declares the details about the identity's principal, including the principal's value, the type (user or service), the relevant configuration property, and a local username mapping. All properties are optional; however, if no base or default value is available (via the parent identity's reference value) for all properties, the principal may be ignored.

The value property of the principal is expected to be the normalized principal name, including the principal's components and realm. In most cases, the realm should be specified using the realm variable (${realm} or ${kerberos-env/realm}). Also, in the case of a service principal, "_HOST" should be used to represent the relevant hostname. This value is typically replaced on the agent side by either the agent-side scripts or the services themselves to be the hostname of the current host. However, the built-in hostname variable (${hostname}) may be used if "_HOST" replacement on the agent-side is not available for the service. Examples: smokeuser@${realm}, service/_HOST@${realm}.

The type property of the principal may be either user or service. If not specified, the type is assumed to be user. This value dictates how the identity is to be created in the KDC or Active Directory. It is especially important in the Active Directory case due to how accounts are created. It also, indicates to Ambari how to handle the principal and relevant keytab file reguarding the user interface behavior and data caching.

The configuration property is an optional configuration specification (config-type/property_name) that is to be set to this principal's value (after its variables have been replaced).

The local_username property, if supplied, indicates which local user account to use when generating auth-to-local rules for this identity. If not specified, no explicit auth-to-local rule will be generated.

```
"principal" : {
  "value": "${cluster-env/smokeuser}@${realm}",
  "type" : "user" ,
  "configuration": "cluster-env/smokeuser_principal_name",
  "local_username" : "${cluster-env/smokeuser}"
}

"principal" : {
  "value": "component1/_HOST@${realm}",
  "type" : "service" ,
  "configuration": "service-site/component1.principal"
}
```

keytab

The keytab block is an optional block inside an identity descriptor block. It describes how to create and store the relevant keytab file. This block declares the keytab file's path in the local filesystem of the destination host, the permissions to assign to that file, and the relevant configuration property.

The file property declares an absolute path to use to store the keytab file when distributing to relevant hosts. If this is not supplied, the keytab file will not be created.

The owner property is an optional block indicating the local user account to assign as the owner of the file and what access ("rw" - read/write; "r" - read-only) should be granted to that user. By default, the owner will be given read-only access.

The group property is an optional block indicating which local group to assigned as the group owner of the file and what access ("rw" - read/write; "r" - read-only; “” - no access) should be granted to local user accounts in that group. By default, the group will be given no access.

The configuration property is an optional configuration specification (config-type/property_name) that is to be set to the path of this keytabs file (after any variables have been replaced).

```
"keytab" : {
  "file": "${keytab_dir}/smokeuser.headless.keytab",
  "owner": {
    "name": "${cluster-env/smokeuser}",
    "access": "r"
  },
  "group": {
    "name": "${cluster-env/user_group}",
    "access": "r"
  },
  "configuration": "${cluster-env/smokeuser_keytab}"
}
```

services

A services block may exist in the stack-level and the service-level Kerberos Descriptor file. This block is a list of zero or more service descriptors to add to the Kerberos Descriptor.

Each service block contains a service name, and optional identities, auth_to_local_properties configurations, and components blocks.

```
"services": [
  {
    "name": "SERVICE1_NAME",
    "identities": [
      ...
    ],
    "auth_to_local_properties" : [
      ...
    ],
    "configurations": [
      ...
    ],
    "components": [
      ...
    ]
  },
  {
    "name": "SERVICE2_NAME",
    "identities": [
      ...
    ],
    "auth_to_local_properties" : [
      ...
    ],
    "configurations": [
      ...
    ],
    "components": [
      ...
    ]
  },
  …
]
```

components

A components block may exist within a service descriptor block. This block is a list of zero or more component descriptors belonging to the containing service descriptor. Each component descriptor is a block containing a component name, and optional identities, auth_to_local_properties, and configurations blocks.

```
"components": [
  {
    "name": "COMPONENT_NAME",
    "identities": [
      ...
    ],
    "auth_to_local_properties" : [
      ...
    ],
    "configurations": [
      ...
```

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

Examples

Example Stack-level Kerberos Descriptor

The following example is annotated for descriptive purposes. The annotations are not valid in a real JSON-formatted file.

```
{
  // Properties that can be used in variable replacement operations.
  // For example, ${keytab_dir} will resolve to "/etc/security/keytabs".
  // Since variable replacement is recursive, ${realm} will resolve
  // to ${kerberos-env/realm}, which in-turn will resolve to the
  // declared default realm for the cluster
  "properties": {
    "realm": "${kerberos-env/realm}",
    "keytab_dir": "/etc/security/keytabs"
  },
  // A list of global Kerberos identities. These may be referenced
  // using /identity_name. For example the “spnego” identity may be
  // referenced using “/spnego”
  "identities": [
    {
      "name": "spnego",
      // Details about this identity's principal. This instance does not
      // declare any value for configuration or local username. That is
      // left up to the services and components use wish to reference
      // this principal and set overrides for those values.
      "principal": {
        "value": "HTTP/_HOST@${realm}",
        "type" : "service"
      },
      // Details about this identity’s keytab file. This keytab file
      // will be created in the configured keytab file directory with
      // read-only access granted to root and users in the cluster’s
      // default user group (typically, hadoop). To ensure that only
      // a single copy exists on the file system, references to this
      // identity should not override the keytab file details;
      // however if it is desired that multiple keytab files are
      // created, these values may be overridden in a reference
      // within a service or component. Since no configuration
      // specification is set, the the keytab file location will not
      // be set in any configuration file by default. Services and
      // components need to reference this identity to update this
      // value as needed.
      "keytab": {
        "file": "${keytab_dir}/spnego.service.keytab",
        "owner": {
          "name": "root",
          "access": "r"
        },
        "group": {
          "name": "${cluster-env/user_group}",
          "access": "r"
        }
      }
    },
```

```
    {
      "name": "smokeuser",
      // Details about this identity's principal. This instance declares
      // a configuration and local username mapping. Services and
      // components can override this to set additional configurations
      // that should be set to this principal value.  Overriding the
      // local username may create undesired behavior since there may be
      // conflicting entries in relevant auth-to-local rule sets.
      "principal": {
        "value": "${cluster-env/smokeuser}@${realm}",
        "type" : "user",
        "configuration": "cluster-env/smokeuser_principal_name",
        "local_username" : "${cluster-env/smokeuser}"
      },
      // Details about this identity's keytab file. This keytab file
      // will be created in the configured keytab file directory with
      // read-only access granted to the configured smoke user
      // (typically ambari-qa) and users in the cluster's default
      // user group (typically hadoop). To ensure that only a single
      // copy exists on the file system, references to this identity
      // should not override the keytab file details; however if it
      // is desired that multiple keytab files are created, these
      // values may be overridden in a reference within a service or
      // component.
      "keytab": {
        "file": "${keytab_dir}/smokeuser.headless.keytab",
        "owner": {
          "name": "${cluster-env/smokeuser}",
          "access": "r"
        },
        "group": {
          "name": "${cluster-env/user_group}",
          "access": "r"
        },
        "configuration": "cluster-env/smokeuser_keytab"
      }
    }
  ]
}
```

Example Service-level Kerberos Descriptor

The following example is annotated for descriptive purposes. The annotations are not valid in a real JSON-formatted file.

```
{
  // One or more services may be listed in a service-level Kerberos
  // Descriptor file
  "services": [
    {
      "name": "SERVICE_1",
      // Service-level identities to be created if this service is installed.
      // Any relevant keytab files will be distributed to hosts with at least
      // one of the components on it.
      "identities": [
        // Service-specific identity declaration, declaring all properties
        // needed initiate the creation of the principal and keytab files,
        // as well as setting the service-specific  configurations.  This may
        // be referenced by contained components using ../service1_identity.
        {
          "name": "service1_identity",
          "principal": {
```

```
      "value": "service1/_HOST@${realm}",
      "type" : "service",
      "configuration": "service1-site/service1.principal"
    },
    "keytab": {
      "file": "${keytab_dir}/service1.service.keytab",
      "owner": {
        "name": "${service1-env/service_user}",
        "access": "r"
      },
      "group": {
        "name": "${cluster-env/user_group}",
        "access": "r"
      },
      "configuration": "service1-site/service1.keytab.file"
    }
  },
  // Service-level identity referencing the stack-level spnego
  // identity and overriding the principal and keytab configuration
  // specifications.
  {
    "name": "service1_spnego",
    "reference": "/spnego",
    "principal": {
      "configuration": "service1-site/service1.web.principal"
    },
    "keytab": {
      "configuration": "service1-site/service1.web.keytab.file"
    }
  },
  // Service-level identity referencing the stack-level smokeuser
  // identity. No properties are being overridden and overriding
  // the principal and keytab configuration specifications.
  // This ensures that the smokeuser principal is created and its
  // keytab file is distributed to all hosts where components of this
  // this service are installed.
  {
    "name": "service1_smokeuser",
    "reference": "/smokeuser"
  }
],
// Properties related to this service that require the auth-to-local
// rules to be dynamically generated based on identities create for
// the cluster.
"auth_to_local_properties" : [
  "service1-site/security.auth_to_local"
],
// Configuration properties to be set when this service is installed,
// no matter which components are installed
"configurations": [
  {
    "service-site": {
      "service1.security.authentication": "kerberos",
      "service1.security.auth_to_local": ""
    }
  }
],
// A list of components related to this service
"components": [
  {
    "name": "COMPONENT_1",
    // Component-specific identities to be created when this component
    // is installed.  Any keytab files specified will be distributed
```

```
          // only to the hosts where this component is installed.
          "identities": [
            // An identity "local" to this component
            {
              "name": "component1_service_identity",
              "principal": {
                "value": "component1/_HOST@${realm}",
                "type" : "service",
                "configuration": "service1-site/comp1.principal",
                "local_username" : "${service1-env/service_user}"
              },
              "keytab": {
                "file": "${keytab_dir}/s1c1.service.keytab",
                "owner": {
                  "name": "${service1-env/service_user}",
                  "access": "r"
                },
                "group": {
                  "name": "${cluster-env/user_group}",
                  "access": ""
                },
                "configuration": "service1-site/comp1.keytab.file"
              }
            },
            // The stack-level spnego identity overridden to set component-specific
            // configurations
            {
              "name": "component1_spnego_1",
              "reference": "/spnego",
              "principal": {
                "configuration": "service1-site/comp1.spnego.principal"
              },
              "keytab": {
                "configuration": "service1-site/comp1.spnego.keytab.file"
              }
            },
            // The stack-level spnego identity overridden to set a different set of
component-specific
            // configurations
            {
              "name": "component1_spnego_2",
              "reference": "/spnego",
              "principal": {
                "configuration": "service1-site/comp1.someother.principal"
              },
              "keytab": {
                "configuration": "service1-site/comp1.someother.keytab.file"
              }
            }
          ],
          // Component-specific configurations to set if this component is installed
          "configurations": [
            {
              "service-site": {
                "comp1.security.type": "kerberos"
              }
            }
          ]
        },
        {
          "name": "COMPONENT_2",
          "identities": [
            {
```

```
            "name": "component2_service_identity",
            "principal": {
              "value": "component2/_HOST@${realm}",
              "type" : "service",
              "configuration": "service1-site/comp2.principal",
              "local_username" : "${service1-env/service_user}"
            },
            "keytab": {
              "file": "${keytab_dir}/s1c2.service.keytab",
              "owner": {
                "name": "${service1-env/service_user}",
                "access": "r"
              },
              "group": {
                "name": "${cluster-env/user_group}",
                "access": ""
              },
              "configuration": "service1-site/comp2.keytab.file"
            }
          },
          // The service-level service1_identity identity overridden to
          // set component-specific configurations
          {
            "name": "component2_service1_identity",
            "reference": "../service1_identity",
            "principal": {
              "configuration": "service1-site/comp2.service.principal"
            },
            "keytab": {
              "configuration": "service1-site/comp2.service.keytab.file"
            }
          }
        ],
        "configurations" : [
          {
            "service-site" : {
              "comp2.security.type": "kerberos"
            }
          }
        ]
      }
    ]
  }
}
```

Metrics

Ambari provides the Ambari Metrics System ("AMS") service for collecting, aggregating and serving Hadoop and system metrics in Ambari-managed clusters.

Each service can define which metrics AMS should collect and provide by defining a metrics.json file.

Metrics Structure

Key	Allowed Values	Comments
Type	"ganglia" / "jmx"	ganglia = fulfilled by Ambari Metrics backend service
Category	"default" / "performance"	This is to group metrics into subsets for better navigability
Metrics	`metricKey : { "metricName": "pointInTime": "temporal": }`	**metricKey** = Key to be used by REST API. This is unique for a service and identifies the requested metric as well as what endpoint to use for serving the data (AMS vs JMX) **metricName** = Name to use for the Metrics Service backend **pointInTime** = Get latest value, no time range query allowed **temporal** = Time range query supported

Example

```
{
  "NAMENODE": {
    "Component": [
      {
        "type": "ganglia",
        "metrics": {
          "default": {
            "metrics/dfs/FSNamesystem/TotalLoad": {
              "metric": "dfs.FSNamesystem.TotalLoad",
              "pointInTime": false,
              "temporal": true
            }
        } ]
    },
    "HostComponent" : [
         { "type" : "ganglia", ... }
         {  "type" : "jmx", .... }
   ]
}
```

For a detailed example, check out the HDFS metrics.json file.

Quick Links

A service can add a list of quick links to the Ambari web UI by adding a quick links JSON file. Ambari server parses the quick links JSON file and provides its content to the Ambari web UI. The UI can calculate quick link URLs based on that information and populate the quick links drop-down list accordingly.

Declaring Quick Links

In order to add quick links, you first must add the quick link configuration into the metainfo.xml for the service.

```
       <services>
           <service>
           <name>YOUR_SERVICE</name>
           <version>1.0</version>
           <quickLinksConfigurations>
               <quickLinksConfiguration>
                   <fileName>quicklinks.json</fileName>
                   <default>true</default>
               </quickLinksConfiguration>
           </quickLinksConfigurations>
```

You should follow convention and call the JSON file quicklinks.json. You don't need to specify the quicklinks directory. By default, it will assume the directory name is quicklinks under the service root directory. For example, for Oozie, the file is OOZIE/quicklinks/quicklinks.json. You should avoid customizing the quicklinks directory name.

A quick link JSON file has two major sections, the "configuration" section for determine the protocol (HTTP vs HTTPS), and the "links" section for meta information of each quick link to be displayed on the Ambari web UI. The JSON file also includes a "name" section at the top that defines the name of the quick links JSON file that server uses for identification.

The Ambari web UI uses information provided in the "configuration" section to determine if the service is running against HTTP or HTTPS. The result is used to construct all quick link URLs defined in the "links" section.

```
{
    "name": "default",
    "description": "default quick links configuration",
    "configuration": {
        "protocol": {
            # type tells the UI which protocol to use if all checks meet.
            # Use https_only or http_only with empty checks section to explicitly
specify the type
            "type":"https",
            "checks":[ # There can be more than one check needed.
                {
                    "property":"yarn.http.policy",
                    # Desired section here either is a specific value for the property
specified
                    # Or whether the property value should exist or not_exist, blank
or not_blank
                    "desired":"HTTPS_ONLY",
```

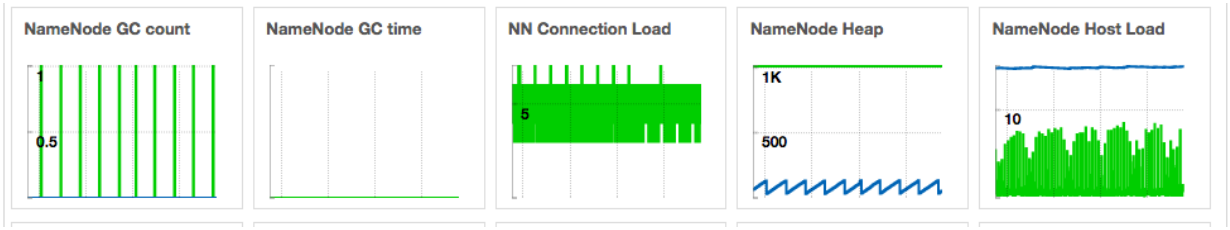
```
                    "site":"yarn-site"
                }
            ]
        },
        #configuration for individual links
        "links": [
            {
                "name": "resourcemanager_ui",
                "label": "ResourceManager UI",
                "requires_user_name": "false", #set this to true if UI should attach
log in user name to the end of the quick link url
                "url": "%@://%@:%@",

                #section calculate the port numbe.
                "port":{
                    #use a property for the whole url if the service does not have a
property for the port.
                    #Specify the regex so the url can be parsed for the port value.
                    "http_property": "yarn.timeline-service.webapp.address",
                    "http_default_port": "8080",
                    "https_property": "yarn.timeline-service.webapp.https.address",
                    "https_default_port": "8090",
                    "regex": "\\w*:(\\d+)",
                    "site": "yarn-site"
                }
            },
            {
                "name": "resourcemanager_logs",
                "label": "ResourceManager logs",
                "requires_user_name": "false",
                "url": "%@://%@:%@/logs",
                "port":{
                    "http_property": "yarn.timeline-service.webapp.address",
                    "http_default_port": "8088",
                    "https_property": "yarn.timeline-service.webapp.https.address",
                    "https_default_port": "8090",
                    "regex": "\\w*:(\\d+)",
                    "site": "yarn-site"
                }
            }
        ]
    }
}
```

Widgets

Each service can define which widgets and heat maps show up by default on the service summary page by defining a widgets.json file. Ambari supports 4 widget types: Graph, Gauge, Number and Template.

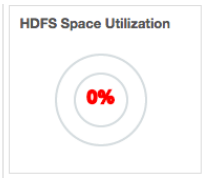
Graph Widget

A widget to display line or area graphs that are derived from one or more than one service metrics value over a range of time.



Gauge Widget

A widget to display percentage calculated from current value of a metric or current values of more than one metric.



Number Widget

A widget to display a number optionally with a unit that can be calculated from the current value of a metric or current values of more than one metric.

Template Widget

A widget to display one or more numbers calculated from the current value of a metric or current values of more than one metric along with an embedded string.

Aggregation

Ambari Metrics System supports 4 type of aggregation for widgets:

```
max: Maximum value of the metric across all host components
min: Minimum value of the metric across all host components
avg: Average value of the metric across all host components
sum: Sum of metric value recorded for each host components
```

By default, the Ambari Metrics System uses the average aggregator function while computing the value for a service component metric but this behavior can be overridden by suffixing the metric name with the aggregator function name. The suffixes are as follows:

- ._max
- ._min
- ._avg
- ._sum

Aggregation Example

The following example, uses the sum aggregator on the hbase regionserver's updatesBlockedTime metric (metrics/hbase/regionserver/Server/updatesBlockedTime).

```
   {
     "name": "regionserver.Server.updatesBlockedTime._sum",
     "metric_path": "metrics/hbase/regionserver/Server/updatesBlockedTime._sum",
     "service_name": "HBASE",
     "component_name": "HBASE_REGIONSERVER"
   }
```

Example Graph Widget

```
{
 "widget_name": "Memory Utilization",
 "description": "Percentage of total memory allocated to containers.",
 "widget_type": "GRAPH",
 "is_visible": true,
 "metrics": [
   {
     "name": "yarn.QueueMetrics.Queue=root.AllocatedMB",
     "metric_path": "metrics/yarn/Queue/root/AllocatedMB",
     "service_name": "YARN",
     "component_name": "RESOURCEMANAGER",
     "host_component_criteria": "host_components/HostRoles/ha_state=ACTIVE"
   },
   {
     "name": "yarn.QueueMetrics.Queue=root.AvailableMB",
     "metric_path": "metrics/yarn/Queue/root/AvailableMB",
     "service_name": "YARN",
     "component_name": "RESOURCEMANAGER",
     "host_component_criteria": "host_components/HostRoles/ha_state=ACTIVE"
   }
 ],
 "values": [
   {
     "name": "Memory Utilization",
     "value": "${(yarn.QueueMetrics.Queue=root.AllocatedMB /
(yarn.QueueMetrics.Queue=root.AllocatedMB + yarn.QueueMetrics.Queue=root.AvailableMB))
* 100}"
   }
 ],
 "properties": {
   "display_unit": "%",
   "graph_type": "LINE",
   "time_range": "1"
 }
}
```

Widget Definition

A widget definition, as seen in the previous example, is made up of the following elements:

- **widget_name**: This is the name that will be displayed in the UI for the widget.
- **description**: Description for the widget that will be displayed in the UI.
- **widget_type**: This information is used by the widget to create the widget from the metric data.
- **is_visible**: This boolean decides if the widget is shown on the service summary page by default or not.

- **metrics**: This is an array that includes all metrics definitions comprising the widget.
- **metrics/name**: Actual name of the metric as being pushed to the sink or emitted as JMX property by the service.
- **metrics/metric_path**: This is the path to which above mentioned metrics/name is mapped in metrics.json file for the service. Metric value will be exposed in the metrics attribute of the service component or host component endpoint of the Ambari API at the same path.
- **metrics/service_name**: Name of the service containing the component emitting the metric.
- **metrics/component_name**: Name of the component emitting the metric.
- **metrics/host_component_criteria:** This is an optional field. Presence of this field means that the metric is host component metric and not a service component metric. If a metric is intended to be queried on host component endpoint then the criteria for choosing the host component needs to be specified over here. If this is left as a single space string then the first found host component will be queried for the metric.

- **values**: This is an array of datasets. Only the Graph widget can have more than one element in the array. All the other widget types always have only one element in the array.
- **values/name**: This field is used only for Graph widget type. This shows up as a label name in the legend for the dataset shown in a Graph widget type.
- **values/value**: This is the expression from which the value for the dataset is calculated. Expression contains references to the declared metric name and constant numbers which acts as valid operands. Expression also contains a valid set of operators {+,-,*,/} that can be used along with valid operands. Parentheses are also permitted in the expression.

- **properties**: These contains set of properties specific to the widget type. For Graph widget type it contains **display_unit**, **graph_type** and **time_range**. The time_range field is currently not honored in the UI.

Service Inheritance

A service can inherit through the stack but may also inherit directly from common-services. This is declared in the metainfo.xml:

```
<metainfo>
  <schemaVersion>2.0</schemaVersion>
  <services>
    <service>
      <name>HDFS</name>
      <extends>common-services/HDFS/2.1.0.2.0</extends>
```

When a service inherits from another service version, how its defining files and directories are inherited follows a number of different patterns.

The following files if defined in the current service version replace the definitions from the parent service version:

```
alerts.json
kerberos.json
metrics.json
role_command_order.json
service_advisor.py
widgets.json
```

Note: All the services' role command orders will be merge with the stack's role command order to provide a master list.

The following files if defined in the current service version are merged with the parent service version (supports removing/deleting parent entries):

```
quicklinks/quicklinks.json
themes/theme.json
```

The following directories if defined in the current service version replace those from the parent service version:

```
packages
upgrades
```

This means the files included in those directories at the parent level will not be inherited. You will need to copy all the files you wish to keep from that directory structure.

The configurations directory in the current service version merges the configuration files with those from the parent service version. Configuration files defined at any level can be omitted from the services configurations by specifying the config-type in the excluded-config-types list:

```
<excluded-config-types>
  <config-type>storm-site</config-type>
</excluded-config-types>
```

For an individual configuration file (or configuration type) like core-site.xml, it will by default merge with the configuration type from the parent. If the `supports_do_not_extend` attribute is specified as `true`, the configuration type will not be merged.

```
<configuration supports_do_not_extend="true">
```

Service MetaInfo Inheritance

By default, all attributes of the service and components if defined in the metainfo.xml of the current service version will replace those of the parent service version unless specified in the sections that follow.

```
<metainfo>
  <schemaVersion>2.0</schemaVersion>
  <services>
    <service>
      <name>HDFS</name>
      <displayName>HDFS</displayName>
      <comment>Apache Hadoop Distributed File System</comment>
      <version>2.1.0.2.0</version>

      <components>
        <component>
          <name>NAMENODE</name>
          <displayName>NameNode</displayName>
          <category>MASTER</category>
          <cardinality>1-2</cardinality>
          <versionAdvertised>true</versionAdvertised>
          <reassignAllowed>true</reassignAllowed>
          ...
```

The custom commands defined in the metainfo.xml of the current service version are merged with those of the parent service version.

```
<customCommands>
  <customCommand>
    <name>DECOMMISSION</name>
    <commandScript>
      <script>scripts/namenode.py</script>
      <scriptType>PYTHON</scriptType>
      <timeout>600</timeout>
    </commandScript>
  </customCommand>
```

The configuration dependencies defined in the metainfo.xml of the current service version are merged with those of the parent service version.

```
<configuration-dependencies>
  <config-type>core-site</config-type>
  <config-type>hdfs-site</config-type>
  ...
</configuration-dependencies>
```

The components defined in the metainfo.xml of the current service are merged with those of the parent (supports delete).

```
<component>
  <name>ZKFC</name>
  <displayName>ZKFailoverController</displayName>
  <category>SLAVE</category>
```

Packaging Custom Services

Management Packs

A management pack is a mechanism for installing stacks, extensions and custom services. A management pack is packaged as a tar.gz file which expands as a directory that includes an mpack.json file and the stack, extension and custom service definitions that it defines.

mpack.json Format

The mpacks.json file allows you to specify the name, version and description of the management pack along with the prerequisites for installing the management pack. For extension management packs, the only important prerequisite is the min_ambari_version. The most important part is the artifacts section. For the purpose here, the artifact type will always be "extension-definitions". You can provide any name for the artifact and you can potentially change the source_dir if you wish to package your extensions under a different directory than "extensions". For consistency, it is recommended that you use the default source_dir "extensions".

```
{
  "type" : "full-release",
  "name" : "myextension-mpack",
  "version": "1.0.0.0",
  "description" : "MyExtension Management Pack",
  "prerequisites": {
    "min_ambari_version" : "2.4.0.0"
  },
  "artifacts": [
    {
      "name" : "myextension-extension-definitions",
      "type" : "extension-definitions",
      "source_dir": "extensions"
    }
  ]
}
```

Extension Management Packs Structure

```
myext-mpack1.0.0.0
├── mpack.json
└── extensions
        └── MY_EXT
                └── 1.0
                      ├── metainfo.xml
                      └── services
                            └── SERVICEA
                └── 2.0
                      ├── metainfo.xml
                      └── services
                            ├── SERVICEA
                            └── …
```

Installing Management Packs

In order to install an extension management pack, you run the following command with or without the "-v" verbose option:

```
ambari-server install-mpack --mpack=/dir/to/myext-mpack-1.0.0.0.tar.gz -v
```

This will check to see if the management pack's prerequisites are met (min_ambari_version). In addition, it will check to see if there are any errors in the management pack format. Assuming everything is correct, the management pack will be extracted in:

```
/var/lib/ambary-server/resources/mpacks
```

It will then create symlinks from /var/lib/ambari-server/resources/extensions for each extension version in /var/lib/ambari-server/resources/mpacks/<mpack dir>/extensions.

Extension Directory	Target Management Pack Symlink
resources/extensions/MY_EXT/1.0	resources/mpacks/myext-mpack1.0.0.0/extensions/MY_EXT/1.0
resources/extensions/MY_EXT/2.0	resources/mpacks/myext-mpack1.0.0.0/extensions/MY_EXT/2.0

Verifying the Extension Installation

Once you have installed the extension management pack, you can restart ambari-server.

```
ambari-server restart
```

After ambari-server has been restarted, you will see in the ambari DB your extension listed in the extension table:

```
ambari=> select * from extension;
extension_id | extension_name | extension_version
--------------+----------------+-------------------
1 | EXT | 1.0
(1 row)
```

You can also query for extensions by calling REST APIs.

```
curl -u admin:admin -H 'X-Requested-By:ambari' -X GET
'http://<server>:<port>/api/v1/extensions'

       {
           "href" : "http://<server>:<port>/api/v1/extensions",
           "items" : [{
               "href" : "http://<server>:<port>/api/v1/extensions/EXT",
               "Extensions" : {
                   "extension_name" : "EXT"
               }
           }]
       }


curl -u admin:admin -H 'X-Requested-By:ambari' -X GET
'http://<server>:<port>/api/v1/extensions/EXT'

       {
           "href" : "http://<server>:<port>/api/v1/extensions/EXT",
           "Extensions" : {
               "extension_name" : "EXT"
           },
           "versions" : [{
               "href" : "http://<server>:<port>/api/v1/extensions/EXT/versions/1.0",
               "Versions" : {
                   "extension_name" : "EXT",
                   "extension_version" : "1.0"
               }
           }]
       }



curl -u admin:admin -H 'X-Requested-By:ambari' -X GET
'http://<server>:<port>/api/v1/extensions/EXT/versions/1.0'

       {
           "href" : "http://<server>:<port>/api/v1/extensions/EXT/versions/1.0",
           "Versions" : {
               "extension-errors" : [ ],
               "extension_name" : "EXT",
               "extension_version" : "1.0",
               "parent_extension_version" : null,
               "valid" : true
           }
       }
```

Linking Extensions to the Stack

Once you have verified that Ambari knows about your extension, the next step is linking the extension version to the current stack version. Linking adds the extension version's services to the list of stack version services. This allows you to install the extension services on the cluster. Linking an extension version to a stack version, will first verify whether the extension supports the given stack version. This is determined by the stack versions listed in the extension version's metainfo.xml.

The following REST API call, will link an extension version to a stack version. In this example it is linking EXT/1.0 with the BIGTOP/1.0 stack version.

```
curl -u admin:admin -H 'X-Requested-By: ambari' -X POST -d '{"ExtensionLink":
{"stack_name": "BIGTOP", "stack_version": "1.0", "extension_name": "EXT",
"extension_version": "1.0"}}' http://<server>:<port>/api/v1/links/
```

You can examine links (or extension links) either in the Ambari DB or with REST API calls.

```
ambari=> select * from extensionlink;
link_id | stack_id | extension_id
---------+----------+--------------
1 | 2 | 1
(1 row)
```

```
curl -u admin:admin -H 'X-Requested-By:ambari' -X GET
'http://<server>:<port>/api/v1/links'
```

```
{
    "href" : "http://<server>:<port>/api/v1/links",
    "items" : [{
        "href" : "http://<server>:<port>/api/v1/links/1",
        "ExtensionLink" : {
            "extension_name" : "EXT",
            "extension_version" : "1.0",
            "link_id" : 1,
            "stack_name" : "BIGTOP",
            "stack_version" : "1.0"
        }
    }]
}
```