Defining a Custom Service

Service Metainfo and Component Category

metainfo.xml

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

Component Category	Default Lifecycle Commands
MASTER	install, start, stop, configure, status
SLAVE	install, start, stop, configure, status
CLIENT	install, configure, status

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
```

Implementing a Custom Service

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

Create a Custom Service

1. Create a directory named **`SAMPLESRV`** that will contain the service definition for **SAMPLESRV**.

   ```
   mkdir SAMPLESRV
   cd SAMPLESRV
   ```

2. 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>
```

3. 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**"
4. Next, let's create that command script. Create a directory for the command script `SAMPLESRV/`**`package/scripts`** that we designated in the service metainfo.

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

5. Within the scripts directory, create the `.py` command script files mentioned in the metainfo.

   For 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()
```

For 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()
```

For 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 a Custom Command

1. 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>
```

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

```
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 Configs to the Custom Service

In this example, we will add a configuration type "test-config" to our SAMPLESRV.

1. Modify the metainfo.xml

   Add the configuration files to the CLIENT component will make it available in the client tar ball downloaded from Ambari.

```
                <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>
```

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

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

3. 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>
```

4. 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>
```

5. 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.

   For example, HIVE defines a component level configuration dependencies for the HIVE_METASTORE component

```
      <component>
          <name>HIVE_METASTORE</name>
          <displayName>Hive Metastore</displayName>
          <category>MASTER</category>
          <cardinality>1</cardinality>
          <versionAdvertised>true</versionAdvertised>
          <reassignAllowed>true</reassignAllowed>
          <clientsToUpdateConfigs></clientsToUpdateConfigs>
... ...
          <configuration-dependencies>
            <config-type>hive-site</config-type>
          </configuration-dependencies>
        </component>
```

   HIVE also defines service level configuration dependencies.

```
    <configuration-dependencies>
       <config-type>core-site</config-type>
       <config-type>hive-log4j</config-type>
       <config-type>hive-exec-log4j</config-type>
       <config-type>hive-env</config-type>
       <config-type>hivemetastore-site.xml</config-type>
       <config-type>webhcat-site</config-type>
       <config-type>webhcat-env</config-type>
       <config-type>parquet-logging</config-type>
       <config-type>ranger-hive-plugin-properties</config-type>
       <config-type>ranger-hive-audit</config-type>
       <config-type>ranger-hive-policymgr-ssl</config-type>
       <config-type>ranger-hive-security</config-type>
       <config-type>mapred-site</config-type>
       <config-type>application.properties</config-type>
       <config-type>druid-common</config-type>
     </configuration-dependencies>
```