

Ambari Stack 和 Service 的增加和删除

滴雨软件科技（上海）有限公司
www.microraindrop.com

介绍

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 extensibility model for new Stacks + Services to be introduced.

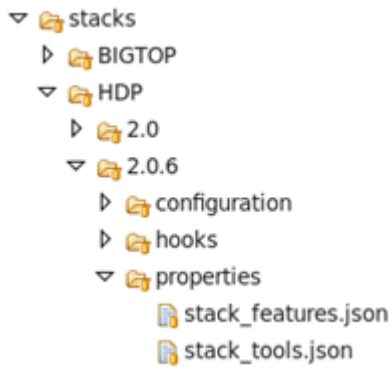
From Ambari 2.4, there is also support for the concept of Extensions and its associated custom Services in an **Extension Definition**.

1.1 Terminology

Term	Description
Stack	定义一组服务，并定义在哪里获取这些服务的软件包。堆栈可具有一个或多个版本，并且每个版本都可以是活动/不活动的。例如，堆栈="HDP-1.3.3"。
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)"

与堆栈配置类似，大多数属性都是在服务级别定义的，但是，堆栈版本级别定义全局属性，可以影响所有服务。一些示例包括：堆栈选择器和配置选择器（[stack-selector](#) and [conf-selector](#)）特定名称或哪些堆栈版本某些堆栈功能受支持。这些属性中的大部分是在 Ambari2.4 版本中引入的，在参数化堆栈信息的努力中，并通过其他分布帮助重用公共服务代码。

在堆栈的“属性”文件夹中，这样的属性以 JSON 格式定义。



1.2 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, Phoenix)...

Stack featurization was added as part of the HDP stack configurations on [HDP/2.0.6/configuration/cluster-env.xml](#), introducing a new `stack_features` property which value is processed in the stack engine from an external property file.

`/HDP/2.0.6/configuration/cluster-env.xml`

```
<!-- Define stack_features property in the base stack. DO NOT override this property for each stack version -->
<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 are defined in [stack_features.json](#) file under `/HDP/2.0.6/properties`. These features support is now available for access at service-level code to change certain service behaviors or configurations. This is an example of features described in `stack_features.json` file:

`stack_features.json`

```
"stack_features": [
  {
    "name": "snappy",
    "description": "Snappy compressor/decompressor support",
    "min_version": "2.0.0.0",
    "max_version": "2.2.0.0"
```

```

    },
    {
        "name": "lzo",
        "description": "LZO libraries support",
        "min_version": "2.2.1.0"
    },
    {
        "name": "express_upgrade",
        "description": "Express upgrade support",
        "min_version": "2.1.0.0"
    },
    {
        "name": "rolling_upgrade",
        "description": "Rolling upgrade support",
        "min_version": "2.2.0.0"
    }
]
}

```

where min_version/max_version are optional constraints.

Feature constants, matching features names, such as `ROLLING_UPGRADE = "rolling_upgrade"` has been added to a new `StackFeature` class in [resource_management/libraries/functions/constants.py](#)

class StackFeature

```

class StackFeature:
    """
    Stack Feature supported
    """
    SNAPPY = "snappy"
    LZ0 = "lzo"
    EXPRESS_UPGRADE = "express_upgrade"
    ROLLING_UPGRADE = "rolling_upgrade"

```

Additionally, corresponding helper functions has been introduced in [resource_management/libraries/functions/stack_features.py](#) to parse the .json file content and called from service code to check if the stack supports the specific feature.

This is an example where the new stack featurization design is used in service code:

```

stack featurization example
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)

```

1.3 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 under /HDP/2.0.6/properties

And declared as part of the HDP stack configurations as a new property on [/HDP/2.0.6/configuration/cluster-env.xml](#)

/HDP/2.0.6/configuration/cluster-env.xml

```
<!-- Define stack_tools property in the base stack. DO NOT override this property for each stack
version -->
<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 [resource_management/libraries/functions/stack_tools.py](#). These helper functions are used to remove hardcodings in resource_management library.

1.4 背景

The Stack definitions can be found in the source tree at [/ambari-server/src/main/resources/stacks](#). After you install the Ambari Server, the Stack definitions can be found at [/var/lib/ambari-server/resources/stacks](#)

1.5 堆栈属性

The stack must contain or inherit a properties directory which contains two files: [stack_features.json](#) and [stack_tools.json](#). The properties directory must be at the root stack version level and must not be included in the other stack versions. This [directory](#) is new in Ambari 2.4.

The stack_features.json contains a list of features that are included in Ambari and allows the stack to specify which versions of the stack include those features. The list of features are determined by the particular Ambari release. The reference list for a particular Ambari version should be found in the [HDP/2.0.6/properties/stack_features.json](#) in the branch for that Ambari release. Each feature has a name and description and the stack can provide the minimum and maximum version where that feature is supported.

```
{
  "stack_features": [
```

```

{
  "name": "snappy",
  "description": "Snappy compressor/decompressor support",
  "min_version": "2.0.0.0",
  "max_version": "2.2.0.0"
},
...
}

```

The `stack_tools.json` includes the name and location where the `stack_selector` and `conf_selector` tools are installed.

```

{
  "stack_selector": ["hdp-select", "/usr/bin/hdp-select", "hdp-select"],
  "conf_selector": ["conf-select", "/usr/bin/conf-select", "conf-select"]
}

```

For more information see the [Stack Properties](#) wiki page.

1.6 结构

The structure of a Stack definition is as follows:

```

|_ stacks
  |_ <stack_name>
    |_ <stack_version>
      metainfo.xml
      |_ hooks
      |_ repos
        repoinfo.xml
      |_ services
        |_ <service_name>
          metainfo.xml
          metrics.json
          |_ configuration
            {configuration files}
          |_ package
            {files, scripts, templates}

```

定义服务和模块

1.1 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 can 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 [here](#).

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

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
```

1.2 利用栈继承

堆栈可以扩展其他堆栈，以便共享命令脚本和配置。这减少了跨堆栈的代码重复，其中包括：

- 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

For example, the **HDP 2.1 Stack extends HDP 2.0.6 Stack** so only the changes applicable to **HDP 2.1 Stack** are present in that Stack definition. This extension is defined in the [metainfo.xml](#) for HDP 2.1 Stack:

```
<metainfo>
  <versions>
    <active>true</active>
  </versions>
  <extends>2.0.6</extends>
</metainfo>
```

1.3 例子：执行客户定制服务

In this example, we will create a custom service called "SAMPLESRV", add it to an existing Stack definition. This service includes MASTER, SLAVE and CLIENT components.

1.3.1 Create and Add the Service

1. On the Ambari Server, browse to the `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services` directory. In this case, we will browse to the HDP 2.0 Stack definition.

```
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services
```

2. Create a directory named `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV` that will contain the service definition for **SAMPLESRV**.

```
mkdir /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV
```

```
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV
```

3. Browse to the newly created 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> <!-- note: use osType rather than
osFamily for Ambari 1.5.0 and 1.5.1 -->
  </osSpecific>
</osSpecifics>
</service>
</services>
</metainfo>

```

4. In the above, my service name is "**SAMPLESRV**", and it contains:

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

5. Next, let's create that command script. Create a directory for the command script `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV/package/scripts` that we designated in the service metainfo.

```
mkdir -p /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV/package/scripts
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/SAMPLESRV/package/scripts
```
6. Browse to the scripts directory and create the `.py` command script files.

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 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';
    def configure(self, env):
        print 'Configure 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 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';
    def configure(self, env):
        print 'Configure 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()
```

7. 重启 Ambari Server, 以便新的服务定义传送到集群中的所有代理。

```
ambari-server restart
```

1.3.2 Install the Service (via Ambari Web "Add Services")

The ability to add custom services via Ambari Web is new as of Ambari 1.7.0.

1. In Ambari Web, browse to Services and click the **Actions** button in the Service navigation area on the left.
2. The "Add Services" wizard launches. You will see an option to include "My Sample Service" (which is the `<displayName>` of the service as defined in the service `metainfo.xml` file).
3. Select "My Sample Service" and click Next.
4. Assign the "Sample Srv Master" and click Next.
5. Select the hosts to install the "Sample Srv Client" and click Next.
6. Once complete, the "My Sample Service" will be available Service navigation area.
7. If you want to add the "Sample Srv Client" to any hosts, you can browse to Hosts and navigate to a specific host and click "+ Add".

1.4 例子：执行定制终端型的服务

In this example, we will create a custom service called "TESTSRV", add it to an existing Stack definition and use the Ambari APIs to install/configure the service. This service is a CLIENT so it has two commands: install and configure.

1.4.1 Create and Add the Service

1. On the Ambari Server, browse to the `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services` directory. In this case, we will browse to the HDP 2.0 Stack definition.
2. Create a directory named `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTSRV` that will contain the service definition for **TESTSRV**.
3. Browse to the newly created TESTSRV 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>TESTSRV</name>
      <displayName>New Test Service</displayName>
      <comment>A New Test Service</comment>
      <version>0.1.0</version>
      <components>
        <component>
          <name>TEST_CLIENT</name>
          <displayName>New Test Client</displayName>
```

```

        <category>CLIENT</category>
        <cardinality>1+</cardinality>
        <commandScript>
            <script>scripts/test_client.py</script>
            <scriptType>PYTHON</scriptType>
            <timeout>600</timeout>
        </commandScript>
        <customCommands>
            <customCommand>
                <name>SOMETHINGCUSTOM</name>
                <commandScript>
                    <script>scripts/test_client.py</script>
                    <scriptType>PYTHON</scriptType>
                    <timeout>600</timeout>
                </commandScript>
            </customCommand>
        </customCommands>
    </component>
</components>
<osSpecifics>
    <osSpecific>
        <osFamily>any</osFamily> <!-- note: use osType rather than
osFamily for Ambari 1.5.0 and 1.5.1 -->
    </osSpecific>
</osSpecifics>
</service>
</services>
</metainfo>

```

4. In the above, my service name is "**TESTSRV**", and it contains one component "**TEST_CLIENT**" that is of component category "**CLIENT**". That client is managed via the command script `scripts/test_client.py`. Next, let's create that command script.
5. Create a directory for the command script `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTSRV/package/scripts` that we designated in the service metainfo.

```

mkdir -p /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTSRV/package/scripts
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTSRV/package/scripts

```

6. Browse to the scripts directory and create the `test_client.py` file. For example:

```

import sys
from resource_management import *

class TestClient(Script):
    def install(self, env):
        print 'Install the client';
    def configure(self, env):
        print 'Configure the client';
    def somethingcustom(self, env):
        print 'Something custom';

if __name__ == "__main__":
    TestClient().execute()

```

7. Now, restart Ambari Server for this new service definition to be distributed to all the Agents in the cluster.

```
ambari-server restart
```

1.4.2 Adding Repository details in repoinfo.xml

When adding a custom service, it may be needed to add additional repository details for the stack especially if the service binaries are available through a separate repository. Additional `<repo>` entries can be added and Ambari will ensure that the base URL provided is used to create repo files on the hosts where service is being installed.

```
<reposinfo>
  <os family="redhat6">
    <repo>
      <baseurl>http://cust.service.lab.com/Services/centos6/1.1/myservices</baseurl>
      <repoid>CUSTOM-1.1</repoid>
      <reponame>CUSTOM</reponame>
    </repo>
    <repo>
      <baseurl>http://public-repo-1.hortonworks.com/HDP/centos6/2.x/updates/2.0.6.1</baseurl>
      <repoid>HDP-2.0.6</repoid>
      <reponame>HDP</reponame>
    </repo>
    <repo>
      <baseurl>http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.17/repos/centos6</baseurl>
      <repoid>HDP-UTILS-1.1.0.17</repoid>
      <reponame>HDP-UTILS</reponame>
    </repo>
  </os>
</reposinfo>
```

1.4.3 Install the Service (via the Ambari REST API)

1. Add the Service to the Cluster.

```
POST
/api/v1/clusters/MyCluster/services
```

```
{
  "ServiceInfo": {
    "service_name": "TESTSRV"
  }
}
```

2. Add the Components to the Service. In this case, add TEST_CLIENT to TESTSRV.

```
POST
/api/v1/clusters/MyCluster/services/TESTSRV/components/TEST_CLIENT
```

3. Install the component on all target hosts. For example, to install on c6402.ambari.apache.org and c6403.ambari.apache.org, first create the host_component resource on the hosts using POST.

```
POST
/api/v1/clusters/MyCluster/hosts/c6402.ambari.apache.org/host_components/TEST_CLIENT
```

```
POST
/api/v1/clusters/MyCluster/hosts/c6403.ambari.apache.org/host_components/TEST_CLIENT
```

4. Now have Ambari install the components on all hosts. In this single command, you are instructing Ambari to install all components related to the service. This call the `install()` method in the command script on each host.

```
PUT
/api/v1/clusters/MyCluster/services/TESTSRV
```

```
{
  "RequestInfo": {
    "context": "Install Test Srv Client"
  },
  "Body": {
    "ServiceInfo": {
      "state": "INSTALLED"
    }
  }
}
```

5. Alternatively, instead of installing all components at the same time, you can explicitly install each host component. In this example, we will explicitly install the TEST_CLIENT on c6402.ambari.apache.org:

```
PUT
/api/v1/clusters/MyCluster/hosts/c6402.ambari.apache.org/host_components/TEST_CLIENT
```

```
{
  "RequestInfo": {
    "context": "Install Test Srv Client"
  },
  "Body": {
    "HostRoles": {
      "state": "INSTALLED"
    }
  }
}
```

6. Use the following to configure the client on the host. This will end up calling the `configure()` method in the command script.

```
POST
/api/v1/clusters/MyCluster/requests
```

```
{
  "RequestInfo" : {
    "command" : "CONFIGURE",
    "context" : "Config Test Srv Client"
  },
  "Requests/resource_filters": [{
    "service_name" : "TESTSRV",
    "component_name" : "TEST_CLIENT",
    "hosts" : "c6403.ambari.apache.org"
  }]
}
```

7. If you want to see which hosts the component is installed.

```
GET
/api/v1/clusters/MyCluster/components/TEST_CLIENT
```

1.4.4 Install the Service (via Ambari Web "Add Services")

The ability to add custom services via Ambari Web is new as of Ambari 1.7.0.

1. In Ambari Web, browse to Services and click the **Actions** button in the Service navigation area on the left.
2. The "Add Services" wizard launches. You will see an option to include "My Test Service" (which is the `<displayName>` of the service as defined in the service `metainfo.xml` file).
3. Select "My Test Service" and click Next.
4. Select the hosts to install the "New Test Client" and click Next.
5. Once complete, the "My Test Service" will be available Service navigation area.
6. If you want to add the "New Test Client" to any hosts, you can browse to Hosts and navigate to a specific host and click "+ Add".

1.5 例子：执行定制终端型的服务（带 Configs）

In this example, we will create a custom service called "TESTCONFIGSRV" and add it to an existing Stack definition. This service is a CLIENT so it has two commands: install and configure. And the service also includes a configuration type "test-config".

Create and Add the Service to the Stack

1. On the Ambari Server, browse to the `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services` directory. In this case, we will browse to the HDP 2.0 Stack definition.

```
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services
```

2. Create a directory named `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV` that will contain the service definition for TESTCONFIGSRV.

```
mkdir /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV
```

```
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV
```

3. Browse to the newly created TESTCONFIGSRV 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>TESTCONFIGSRV</name>
      <displayName>New Test Config Service</displayName>
      <comment>A New Test Config Service</comment>
      <version>0.1.0</version>
      <components>
        <component>
          <name>TESTCONFIG_CLIENT</name>
          <displayName>New Test Config Client</displayName>
          <category>CLIENT</category>
          <cardinality>1+</cardinality>
          <commandScript>
            <script>scripts/test_client.py</script>
            <scriptType>PYTHON</scriptType>
            <timeout>600</timeout>
```

```

        </commandScript>
    </component>
</components>
<osSpecifics>
    <osSpecific>
        <osFamily>any</osFamily> <!-- note: use osType rather than
osFamily for Ambari 1.5.0 and 1.5.1 -->
    </osSpecific>
</osSpecifics>
</service>
</services>
</metainfo>

```

4. In the above, my service name is "**TESTCONFIGSRV**", and it contains one component "**TESTCONFIG_CLIENT**" that is of component category "**CLIENT**". That client is managed via the command script `scripts/test_client.py`. Next, let's create that command script.
5. Create a directory for the command script `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/package/scripts` that we designated in the service metainfo `<commandScript>`.

```

mkdir -p /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/package/scripts
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/package/scripts

```

6. Browse to the scripts directory and create the `test_client.py` file. For example:

```

import sys
from resource_management import *

class TestClient(Script):
    def install(self, env):
        print 'Install the config client';
    def configure(self, env):
        print 'Configure the config client';

if __name__ == "__main__":
    TestClient().execute()

```

7. Now let's define a config type for this service. Create a directory for the configuration dictionary file `/var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/configuration`.

```

mkdir -p /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/configuration
cd /var/lib/ambari-server/resources/stacks/HDP/2.0.6/services/TESTCONFIGSRV/configuration

```

8. Browse to the configuration directory and 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 kool description.</description>
    </property>
</configuration>

```

9. Now, restart Ambari Server for this new service definition to be distributed to all the Agents in the cluster.

ambari-server restart

Stacks and Services

1.1 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 extensibility model for new Stacks + Services to be introduced.

From Ambari 2.4, there is also support for the concept of Extensions and its associated custom Services in an **Extension Definition**.

1.1.1 Terminology

Term	Description
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-1.3.3".
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)"

1.2 Service Metainfo and Component Category

1.2.1 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

```

1.3 Implementing a Custom Service

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

1.3.1 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>
```

```

        </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()

```

1.3.2 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()

```

1.3.3 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 定义了 HIVE_METASTORE 模块在模块级别上的配置依赖。

```
<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>
```

6. HIVE 同时定义了服务级别的配置依赖

```
<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>
```

定制服务

1.1 Introduction

Custom services in Apache Ambari can be packaged and installed in many ways. Ideally, they should all be packaged and installed in the same manner. This document describes how to package and install custom services using Extensions and Management Packs. Using this approach, the custom service definitions do not get inserted under the stack versions services directory. This keeps the stack clean and allows users to easily see which services were installed by which package (stack or extension).

1.2 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.

1.2.1 Example Structure

```
myext-mpack1.0.0.0
├── mpack.json
└── <contents>
```

1.2.2 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"
  }
]
}

```

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 folder to distinguish different releases of the extension which go in the resources/extensions folder with <name>/<version> sub-folders.

扩展的版本类似于堆栈的版本，但是只包括 metainfo.xml 和 the services 目录。 This means that the alerts, kerberos, metrics, role command order and widgets files 不支持，同时这些应该 包括在服务的级别。 In addition, the repositories, hooks, configurations, and upgrades directories are not supported although upgrade support can be added at the service level.

1.1 Extension Structure

```

MY_EXT
├── 1.0
│   ├── metainfo.xml
│   └── services
│       ├── SERVICEA
│       └── ...

```

1.2 Extension metainfo.xml Format:

The extension metainfo.xml is very simple, it just specifies the minimum stack versions which are supported.

```

<metainfo>
  <prerequisites>
    <min-stack-versions>
      <stack>
        <name>BIGTOP</name>

```

```
<version>1.0.*</version>

</stack>

</min-stack-versions>

</prerequisites>

</metainfo>
```

1.3 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:

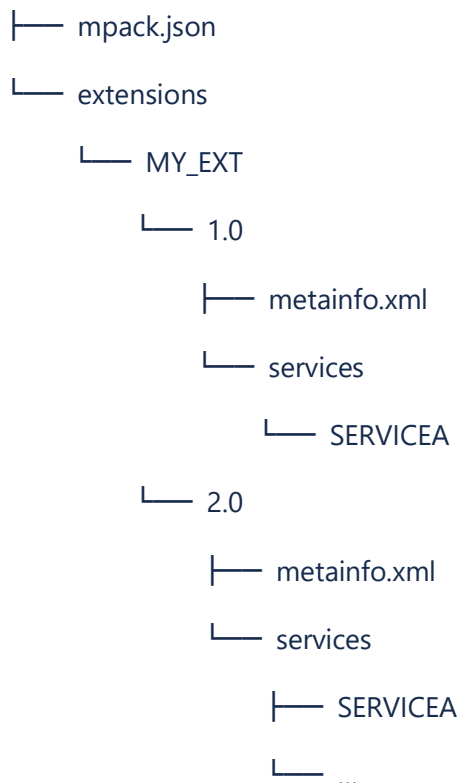
- add new Services in the child Extension version (not in the parent Extension version)
- override command scripts of the parent Services
- override configurations of the parent Services

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>
```

1.4 Extension Management Packs Structure

myext-mpack1.0.0.0



1.5 Installing Management Packs

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

1.6 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
  }
}

```

1.7 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"  
    }  
  }  
}]  
}
```

命令管理

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.

1.1 Sections

Ambari uses the below sections only:

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

1.2 Commands

Commands currently supported by Ambari are

- `INSTALL`
- `UNINSTALL`
- `START`
- `RESTART`
- `STOP`
- `EXECUTE`

- ABORT
- UPGRADE
- SERVICE_CHECK
- CUSTOM_COMMAND
- ACTIONEXECUTE

For more details on role command order, see the [Stack's Role Command Order](#) page.

服务建议

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 which only applies to the services within the stack.

1.1 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.

Sample service-advisor.py file inheritance

```
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):
```

1.2 Service Advisor Behavior

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

1. Recommend layout of the service on cluster
2. Recommend service configurations
3. Validate layout of the service on cluster
4. 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.

Base service_advisor.py from resources/stacks

```
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 []
```

1.3 Examples

- [Service Advisor interface](#)
- [HAWQ 2.0.0 Service Advisor implementation](#)
- [PXF 3.0.0 Service Advisor implementation](#)

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">
```

Inheritance and the Service MetaInfo

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>

          <commandScript>

            <script>scripts/namenode.py</script>

            <scriptType>PYTHON</scriptType>

            <timeout>1800</timeout>

          </commandScript>

          ...

        </component>

      </components>

    </service>

  </services>

</metainfo>
```

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>

</customCommands>
```

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>

```

Each custom service can define its upgrade within its service definition. This allows the custom service to be integrated within the [stack's upgrade](#).

1.1 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.

1.1.1 Examples

- [Upgrades folder](#)
- [Upgrade-pack XML](#)

1.2 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.

1.2.1 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.

1.2.1.1 Prerequisite Checks

The service is able to add its own prerequisite checks.

General Attributes and 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>
```

1.2.1.2 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 Section - Add After Group Entry

```
<order>
  <group xsi:type="cluster" name="PRE_CLUSTER" title="Pre {{direction.text.properties}}>
    <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.

Order Section - Further Add After Group Entry Examples

```
<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.

Order Section - Add After Group

```
<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.

Order Section - Add After Group

```
<group name="F00" title="Foo">
  <add-after-group>HIVE</add-after-group>
  <add-after-group-entry>F00</add-after-group-entry>
  <skippable>true</skippable>
  <allow-retry>false</allow-retry>
  <service name="F002">
    <component>BAR2</component>
  </service>
</group>
```

1.2.1.3 Processing Section

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

Processing Section

```
<processing>
  <service name="F00">
    <component name="BAR">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
    <component name="BAR2">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
  </service>
</processing>
```

Custom Service Repo

Each service can define its own repo info by adding repos/repoinfo.xml in its service folder. The service specific repo will be included in the list of repos defined for the stack.

Example: https://github.com/apache/ambari/blob/trunk/contrib/management-packs/microsoft-r_mpack/src/main/resources/custom-services/MICROSOFT_R_SERVER/8.0.5/repos/repoinfo.xml

```
<reposinfo>
  <os family="redhat6">
    <repo>
      <baseurl>http://cust.service.lab.com/Services/centos6/1.1/myservices</base
      <repoid>CUSTOM-1.1</repoid>
      <reponame>CUSTOM</reponame>
    </repo>
  </os>
</reposinfo>
```

Custom Services - Additional Configuration

1.1 Alerts

Each service is capable of defining which alerts Ambari should track by providing an [alerts.json](#) file.

Read more about Ambari Alerts framework [in the Alerts wiki page](#) and the alerts.json format in the [Alerts definition documentation](#).

1.2 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.

Read more about Kerberos support in the [Automated Kerberization](#) wiki page and the Kerberos descriptor in the [Kerberos Descriptor documentation](#).

1.3 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. Read more about the metrics.json file format in the [Stack Defined Metrics](#) page.

1.4 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.

Read more about quick links JSON file design in the [Quick Links](#) page.

1.5 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.

Read more about the widget descriptors in the [Enhanced Service Dashboard](#) page.

Extension

- [Structure](#)
- [Extension Inheritance](#)
- [Supported Stack Versions](#)
- [Installing Extensions](#)
- [Extension REST APIs](#)
 - [Get all extensions](#)
 - [Get extension](#)
 - [Get extension version](#)
- [Extension Links](#)
- [Extension Link REST APIs](#)
 - [Create Extension Link](#)
 - [Get All Extension Links](#)
 - [Get Extension Link](#)
 - [Delete Extension Link](#)
 - [Update All Extension Links](#)

1.1 Background

Added in Ambari 2.4.

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.

1.2 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.

1.3 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>
```

1.4 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>
```

1.5 Installing Extensions

It is recommended to install extensions using [management packs](#). For more details see the [instructions on packaging custom services using extensions and management packs](#).

Once the extension version directory has been created under the `resource/extensions` directory with the required `metainfo.xml` file, you can restart `ambari-server`.

1.6 Extension REST APIs

You can query for extensions by calling REST APIs.

1.6.1 Get all extensions

```
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"
      }
    }
  ]
}
```

1.6.2 Get extension

```
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"
      }
    }
  ]
}
```

1.6.3Get extension version

```
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
  }
}
```

1.6.4Extension 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.

1.7 Extension Link REST APIs

You can retrieve, create, update and delete extension links by calling REST APIs.

1.7.1Create Extension Link

The following curl command will link an extension EXT/1.0 to the stack HDP/2.4

```
curl -u admin:admin -H 'X-Requested-By: ambari' -X POST -d '{"ExtensionLink": {"st
"2.4", "extension_name": "EXT", "extension_version": "1.0"}}' http://<server>:<por
```

1.7.2Get All Extension Links

```
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>:8080/api/v1/links/1",
    "ExtensionLink" : {
      "extension_name" : "EXT",
      "extension_version" : "1.0",
      "link_id" : 1,
      "stack_name" : "HDP",
      "stack_version" : "2.4"
    }
  }
]
}

```

1.7.3 Get Extension Link

```

curl -u admin:admin -H 'X-Requested-By:ambari' -X GET 'http://<server>:<port>/api/v1/link/1'
{
  "href" : "http://<server>:<port>/api/v1/links/1",
  "ExtensionLink" : {
    "extension_name" : "EXT",
    "extension_version" : "1.0",
    "link_id" : 1,
    "stack_name" : "HDP",
    "stack_version" : "2.4"
  }
}

```

1.7.4 Delete Extension Link

You must specify the ID of the Extension Link to be deleted.

```

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

```

1.7.5 Update All Extension Links

This will reread the stacks, extensions and services in order to make sure the state of the stack is up to date in memory.

```

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

```


How-To Define Stacks and Services

Services managed by Ambari are defined in its *stacks* folder.

To define your own services and stacks to be managed by Ambari, follow the steps below. There is also an example you can follow on how to [create your custom stack and service](#).

A stack is a collection of services. Multiple versions of a stack can be defined, each with its own set of services. Stacks in Ambari are defined in [ambari-server/src/main/resources/stacks](#) folder, which can be found at `/var/lib/ambari-server/resources/stacks` folder after install.

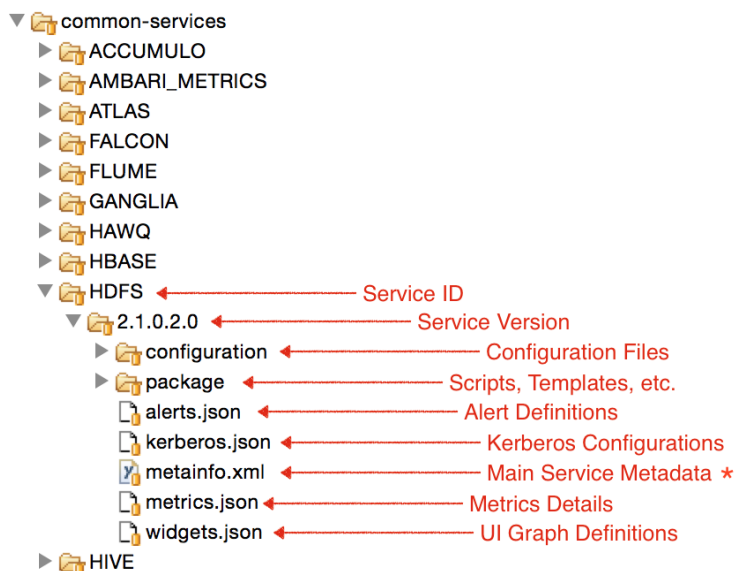
Services managed by a stack can be defined either in [ambari-server/src/main/resources/common-services](#) or [ambari-server/src/main/resources/stacks](#) folders. These folders after install can be found at `/var/lib/ambari-server/resources/common-services` or `/var/lib/ambari-server/resources/stacks` folders respectively.

Question: When do I define service in *common-services* vs. *stacks* folders?

One would define services in the [common-services](#) folder if there is possibility of the service being used in multiple stacks. For example, almost all stacks would need the HDFS service - so instead of redefining HDFS in each stack, the one defined in **common-services is referenced**. Likewise, if a service is never going to be shared, it can be defined in the [stacks](#) folder. Basically services defined in stacks folder are used by containment, whereas the ones defined in common-services are used by reference.

1.1 Define Service

Shown below is how to define a service in *common-services* folder. The same approach can be taken when defining services in the *stacks* folder, which will be discussed in the *Define Stack* section.



Services **MUST** provide the main *metainfo.xml* file which provides important metadata about the service. Apart from that, other files can be provided to give more information about the service. More details about these files are provided below.

A service may also inherit from either a previous stack version or common services. For more information see the [Service Inheritance](#) page.

1.2 *metainfo.xml*

In the *metainfo.xml* service descriptor, one can first define the service and its components.

Complete reference can be found in the [Writing metainfo.xml](#) page.

A good reference implementation is the [HDFS metainfo.xml](#).

Question: Is it possible to define multiple services in the same metainfo.xml?

Yes. Though it is possible, it is discouraged to define multiple services in the same service folder.

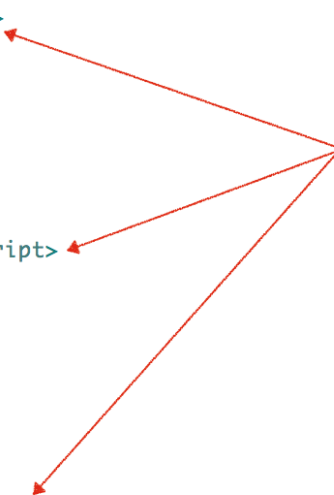
YARN and MapReduce2 are services that are defined together in the [YARN folder](#). Its [metainfo.xml](#) defines both services.

1.2.1 Scripts

With the components defined, we need to provide scripts which can handle the various stages of the service and component's lifecycle.

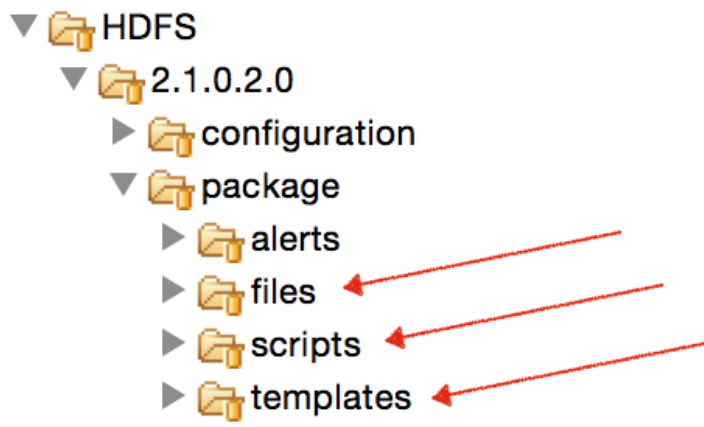
The scripts necessary to manage service and components are specified in the *metainfo.xml* ([HDFS](#)) Each of these scripts should extend the [Script](#) class which provides useful methods. Example: [NameNode script](#)

```
<components>
  <component>
    <name>NAMENODE</name>
    <displayName>NameNode</displayName>
    <category>MASTER</category>
    <cardinality>1-2</cardinality>
    <versionAdvertised>true</versionAdvertised>
    <commandScript>
      <script>scripts/namenode.py</script>
      <scriptType>PYTHON</scriptType>
      <timeout>1800</timeout>
    </commandScript>
    <customCommands>
      <customCommand>
        <name>DECOMMISSION</name>
        <commandScript>
          <script>scripts/namenode.py</script>
          <scriptType>PYTHON</scriptType>
          <timeout>600</timeout>
        </commandScript>
      </customCommand>
      <customCommand>
        <name>REBALANCEHDFS</name>
        <background>true</background>
        <commandScript>
          <script>scripts/namenode.py</script>
          <scriptType>PYTHON</scriptType>
        </commandScript>
      </customCommand>
    </customCommands>
  </component>
```



Reference to scripts used by NameNode

These scripts should be provided in the `<service-id>/<service-version>/package/scripts` folder.



package/scripts	Contains scripts invoked by Ambari. These scripts are loaded into the execution path with the correct environment. Example: HDFS
package/files	Contains files used by above scripts. Generally these are other scripts (bash, python, etc.) invoked as a separate process. Example: checkWebUI.py is run in HDFS service-check to determine if Journal Nodes are available
package/templates	Template files used by above scripts to generate files on managed hosts. These are generally configuration files required by the service to operate. Example: exclude_hosts_list.j2 which is used by scripts to generate <code>/etc/hadoop/conf/dfs.exclude</code>

1.2.2Python

Ambari by default supports Python scripts for management of service and components.

Component scripts should extend `resource_management.Script` class and provide methods required for that component's lifecycle.

Taken from the page on [how to create custom stack](#), the following methods are needed for MASTER, SLAVE and CLIENT components to go through their lifecycle.

master.py

```

1  import sys
2  from resource_management import Script
3  class Master(Script):
4      def install(self, env):
5          print 'Install the Sample Srv Master';
6      def stop(self, env):
7          print 'Stop the Sample Srv Master';
8      def start(self, env):
9          print 'Start the Sample Srv Master';
10     def status(self, env):
11         print 'Status of the Sample Srv Master';
12     def configure(self, env):
13         print 'Configure the Sample Srv Master';
14     if __name__ == "__main__":
15         Master().execute()
  
```

slave.py

```

1  import sys
2  from resource_management import Script
  
```



```

3     class Slave(Script):
4         def install(self, env):
5             print 'Install the Sample Srv Slave';
6         def stop(self, env):
7             print 'Stop the Sample Srv Slave';
8         def start(self, env):
9             print 'Start the Sample Srv Slave';
10        def status(self, env):
11            print 'Status of the Sample Srv Slave';
12        def configure(self, env):
13            print 'Configure the Sample Srv Slave';
14    if __name__ == "__main__":
15        Slave().execute()

```

client.py

```

1     import sys
2     from resource_management import Script
3     class SampleClient(Script):
4         def install(self, env):
5             print 'Install the Sample Srv Client';
6         def configure(self, env):
7             print 'Configure the Sample Srv Client';
8     if __name__ == "__main__":
9         SampleClient().execute()

```

Ambari provides helpful Python libraries below which are useful in writing service scripts. For complete reference on these libraries visit the [Ambari Python Libraries](#) page.

- resource_management
- ambari_commons
- ambari_simplejson

1.2.2.1 OS Variant Scripts

If the service is supported on multiple OSes which requires separate scripts, the base `resource_management.Script` class can be extended with different `@OsFamilyImpl()` annotations. This allows for the separation of only OS specific methods of the component.

Example: [NameNode default script](#), [NameNode Windows script](#).

Examples

NameNode [Start](#), [Stop](#).

DataNode [Start and Stop](#).

HDFS [configurations persistence](#)

1.2.3 Custom Actions

Sometimes services need to perform actions unique to that service which go beyond the default actions provided by Ambari (like *install*, *start*, *stop*, *configure*, etc.).

Services can define such actions and expose them to the user in UI so that they can be easily invoked.

As an example, we show the *Rebalance HDFS* custom action implemented by HDFS.

1.2.3.1 Stack Changes

1. [Define custom command inside the *customCommands* section](#) of the component in *metainfo.xml*.
2. [Implement method with same name as custom command](#) in script referenced from *metainfo.xml*.
 - a. If custom command does not have OS variants, it can be implemented in the same class that extends *resource_management.Script*
 - b. If there are OS variants, different methods can be implemented in each class annotated by `@OsFamilyImpl(os_family=...)`. [Default rebalancehdfs](#), [Windows rebalancehdfs](#).

This will provide ability by the backend to run the script on all managed hosts where the service is installed.

1.2.3.2 UI Changes

No UI changes are necessary to see the custom action on the host page. The action should show up in the host-component's list of actions. Any master-component actions will automatically show up on the service's action menu.

When the action is clicked in UI, the POST call is made automatically to trigger the script defined above.

Question: How do I provide my own label and icon for the custom action in UI?

In Ambari UI, add your component action to the *App.HostComponentActionMap* object with custom icon and name. Ex: [REBALANCEHDFS](#).

1.1 Configuration

Configuration files for a service should be placed by default in the [configuration](#) folder. If a different named folder has to be used, the [configuration-dir](#) element can be used in *metainfo.xml* to point to that folder.

The important sections of the metainfo.xml with regards to configurations are:

```
<?xml version="1.0"?>
<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>HDFS_CLIENT</name>
          ...
          <configFiles>
            <configFile>
              <type>xml</type>
              <fileName>hdfs-site.xml</fileName>
              <dictionaryName>hdfs-site</dictionaryName>
            </configFile>
            <configFile>
              <type>xml</type>
              <fileName>core-site.xml</fileName>
```

```

        <dictionaryName>core-site</dictionaryName>
    </configFile>
</configFile>
    <type>env</type>
    <fileName>log4j.properties</fileName>
    <dictionaryName>hdfs-log4j,yarn-log4j</dictionaryName>
</configFile>
</configFile>
    <type>env</type>
    <fileName>hadoop-env.sh</fileName>
    <dictionaryName>hadoop-env</dictionaryName>
</configFile>
</configFiles>
...
<configuration-dependencies>
    <config-type>core-site</config-type>
    <config-type>hdfs-site</config-type>
</configuration-dependencies>
</component>
...
</components>

<configuration-dir>configuration</configuration-dir>
<configuration-dependencies>
    <config-type>core-site</config-type>
    <config-type>hdfs-site</config-type>
    <config-type>hadoop-env</config-type>
    <config-type>hadoop-policy</config-type>
    <config-type>hdfs-log4j</config-type>
    <config-type>ranger-hdfs-plugin-properties</config-type>
    <config-type>ssl-client</config-type>
    <config-type>ssl-server</config-type>
    <config-type>ranger-hdfs-audit</config-type>
    <config-type>ranger-hdfs-policymgr-ssl</config-type>
    <config-type>ranger-hdfs-security</config-type>
    <config-type>ams-ssl-client</config-type>
</configuration-dependencies>
</service>
</services>
</metainfo>

```

- **config-type** - String representing a group of configurations. Example: *core-site*, *hdfs-site*, *yarn-site*, etc. When configurations are saved in Ambari, they are persisted within a version of config-type which is immutable. If you change and save HDFS core-site configs 4 times, you will have 4 versions of config-type core-site. Also, when a service's configs are saved, only the changed config-types are updated.
 - **configFiles** - lists the config-files handled by the enclosing component
 - **configFile** - represents one config-file of a certain type
 - **type** - type of file based on which contents are generated differently
 - **xml** - XML file generated in Hadoop friendly format. Ex: [hdfs-site.xml](#)
 - **env** - Generally used for scripts where the content value is used as a template. The template has config-tags whose values are populated at runtime during file generation. Ex: [hadoop-env.sh](#)

- **properties** - Generates property files where entries are in key=value format. Ex: [falcon-runtime.properties](#)
- **dictionaryName** - Name of the config-type as which key/values of this config file will be stored
- **configuration-dependencies** - Lists the config-types on which this component or service depends on. 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. From the code section above, whenever *core-site* is updated, both HDFS service as well as HDFS_CLIENT component will be marked as requiring restart.
- **configuration-dir** - Directory where files listed in *configFiles* will be. Optional. Default value is *configuration*.

1.1.1 Adding new configs in a config-type

There are a number of different parameters that can be specified to a config item when it is added to a config-type. These have been covered [here](#).

1.1.2 UI - Categories

Configurations defined above show up in the service's *Configs* page.

To customize categories and ordering of configurations in UI, the following files have to be updated.

Create Category - Update the [ambari-web/app/models/stack_service.js](#) file to add your own service, along with your new categories.

Use Category - To place configs inside a defined category, and specify an order in which configs are placed, add configs to [ambari-web/app/data/HDP2/site_properties.js](#) file. In this file one can specify the category to use, and the index where a config should be placed. The stack folders in [ambari-web/app/data](#) are hierarchical and inherit from previous versions. The mapping of configurations into sections is defined here. Example [Hive Categories](#), [Tez Categories](#).

1.1.3 UI - Enhanced Configs

The *Enhanced Configs* feature makes it possible for service providers to customize their service's configs to a great deal and determine which configs are prominently shown to user without making any UI code changes. Customization includes providing a service friendly layout, better controls (sliders, combos, lists, toggles, spinners, etc.), better validation (minimum, maximum, enums), automatic unit conversion (MB, GB, seconds, milliseconds, etc.), configuration dependencies and improved dynamic recommendations of default values.

A service provider can accomplish all the above just by changing their service definition in the *stacks/* folder.

Read more in the [Enhanced Configs](#) page

1.2 Alerts

Each service is capable of defining which alerts Ambari should track by providing an [alerts.json](#) file.

Read more about Ambari Alerts framework [in the Alerts wiki page](#) and the alerts.json format in the [Alerts definition documentation](#).

1.3 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.

Read more about Kerberos support in the [Automated Kerberization](#) wiki page and the Kerberos descriptor in the [Kerberos Descriptor documentation](#).

1.4 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. You can read about the metrics.json file format in the [Stack Defined Metrics](#) page.

1.5 Quick Links

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

Read more about quick links JSON file design in the [Quick Links](#) page.

1.6 Widgets

Each service can define which widgets and heatmaps show up by default on the service summary page by defining a [widgets.json](#) file.

You can read more about the widgets descriptor in the [Enhanced Service Dashboard](#) page.

1.7 Role Command Order

From Ambari 2.2, each service can define its own role command order by including the [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:

Example service role_command_order.json

```
"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 in addition the COMP_X-START dependency on NAMENODE-START would just be added as a new dependency.

For more details on role command order, see the [Role Command Order](#) section below.

1.8 Service Advisor

From Ambari 2.4, each service can choose to define its own service advisor rather than define the details of its configuration and layout in the stack advisor. This is particularly useful for custom services which are not defined in the stack. Ambari provides the *Service Advisor* capability where a service can write a Python script named *service-advisor.py* in their service folder. This folder can be in the stack's services directory where the

service is defined or can be inherited from the service definition in common-services or elsewhere. Example: [common-services/HAWQ/2.0.0](#).

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.

Sample service-advisor.py file inheritance

```
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',))
except Exception as e:
    traceback.print_exc()
    print "Failed to load parent"

class HAWQ200ServiceAdvisor(service_advisor.ServiceAdvisor):
```

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

1. Recommend layout of the service on cluster
2. Recommend service configurations
3. Validate layout of the service on cluster
4. 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.

Base service_advisor.py from resources/stacks

```
class ServiceAdvisor(DefaultStackAdvisor):
    """
    Abstract class implemented by all service advisors.
    """

    """
    If any components of the service should be colocated with other services,
    this is where you should set up that layout. Example:

    # colocate HAWQSEGMENT with DATANODE, if no hosts have been allocated for HAWQSEGMENT
    hawqSegment = [component for component in serviceComponents if component["StackServiceC
    if not self.isComponentHostsPopulated(hawqSegment):
        for hostName in hostsComponentsMap.keys():
            hostComponents = hostsComponentsMap[hostName]
            if {"name": "DATANODE"} in hostComponents and {"name": "HAWQSEGMENT"} not in hostCor
                hostsComponentsMap[hostName].append( { "name": "HAWQSEGMENT" } )
            if {"name": "DATANODE"} not in hostComponents and {"name": "HAWQSEGMENT"} in hostCor
                hostComponents.remove({"name": "HAWQSEGMENT"})
    """
```

```

def colocateService(self, hostsComponentsMap, serviceComponents):
    pass

"""
Any configuration recommendations for the service should be defined in this function.
This should be similar to any of the recommendXXXXConfigurations functions in the stack_advisor.py
such as recommendYARNConfigurations().
"""

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

"""
Returns an array of Validation objects about issues with the hostnames to which components are placed.
This should detect validation issues which are different than those the stack_advisor.py
The default validations are in stack_advisor.py getComponentLayoutValidations function.
"""

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

"""
Any configuration validations for the service should be defined in this function.
This should be similar to any of the validateXXXXConfigurations functions in the stack_advisor.py
such as validateHDFSConfigurations.
"""

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

```

1.8.1.1 Examples

- [Service Advisor interface](#)
- [HAWQ 2.0.0 Service Advisor implementation](#)
- [PXF 3.0.0 Service Advisor implementation](#)

1.9 Service Upgrade

From Ambari 2.4, each service can now define its upgrade within its service definition. This is particularly useful for custom services which no longer need to modify the stack's upgrade-packs in order to integrate themselves into the [cluster upgrade](#).

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.

1.9.1 Examples

- [Upgrades folder](#)
- [Upgrade-pack XML](#)

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.

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. The service is able to add its own prerequisite checks.

General Attributes and 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>
```

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 Section - Add After Group Entry

```
<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.

Order Section - Further Add After Group Entry Examples

```
<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.

Order Section - Add After Group

```
<group name="F00" title="Foo">
  <add-after-group>HIVE</add-after-group>
  <skippable>true</skippable>
  <allow-retry>false</allow-retry>
  <service name="F00">
    <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.

Order Section - Add After Group

```
<group name="F00" title="Foo">
  <add-after-group>HIVE</add-after-group>
  <add-after-group-entry>F00</add-after-group-entry>
  <skippable>true</skippable>
  <allow-retry>false</allow-retry>
  <service name="F002">
    <component>BAR2</component>
  </service>
</group>
```

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

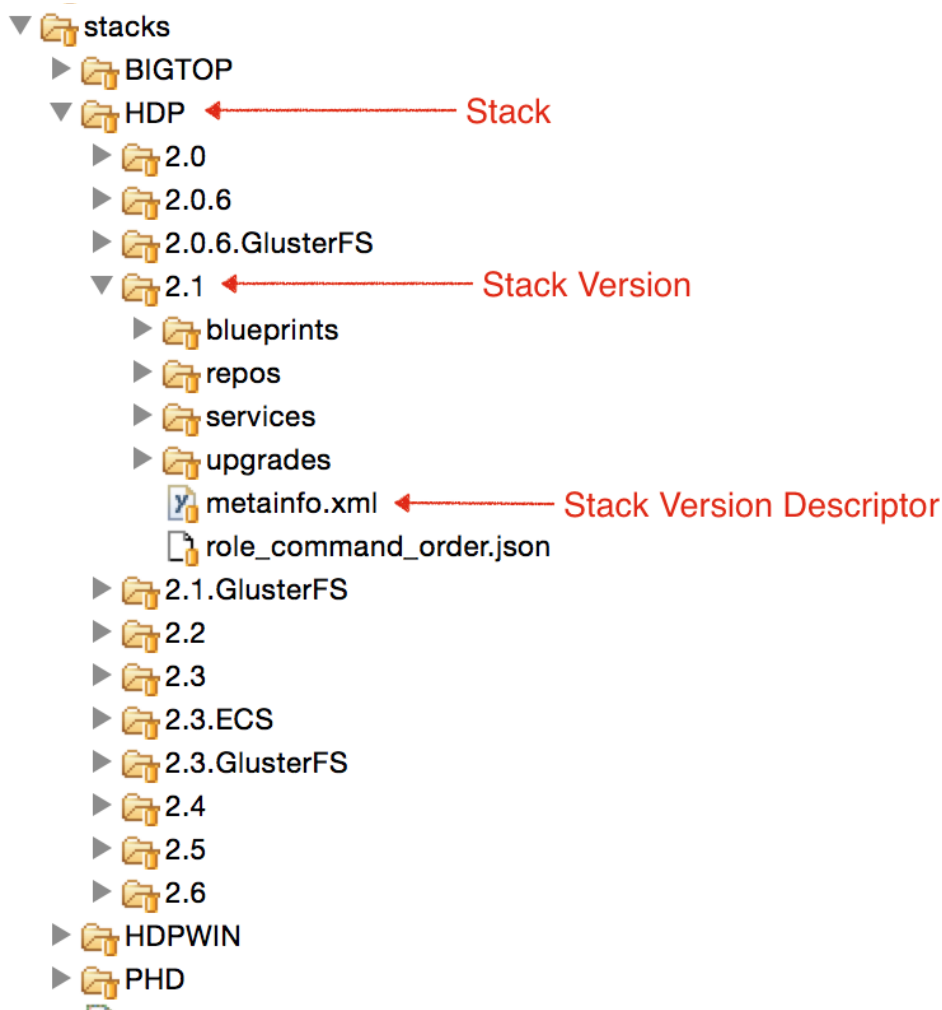
Processing Section

```
<processing>
  <service name="F00">
    <component name="BAR">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
    <component name="BAR2">
      <upgrade>
        <task xsi:type="restart-task" />
      </upgrade>
    </component>
  </service>
</processing>
```

1.10 Define Stack

A stack is a versioned collection of services. Each stack is a folder is defined in [ambari-server/src/main/resources/stacks](#) source. Once installed, these stack definitions are available on the ambari-server machine at `/var/lib/ambari-server/resources/stacks`.

Each stack folder contains one sub-folder per version of the stack. Some of these stack-versions are active while some are not. Each stack-version includes services which are either referenced from *common-services*, or defined inside the stack-version's *services* folder.



Example: [HDP stack](#), [HDP-2.4 stack version](#).

1.11 Stack-Version Descriptor

Each stack-version should provide a *metainfo.xml* (Example: [HDP-2.3](#), [HDP-2.4](#)) 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.

1.12 Stack Properties

The stack must contain or inherit a properties directory which contains two files: [stack_features.json](#) and [stack_tools.json](#). This [directory](#) is new in Ambari 2.4.

The `stack_features.json` contains a list of features that are included in Ambari and allows the stack to specify which versions of the stack include those features. The list of features are determined by the particular Ambari release. The reference list for a particular Ambari version should be found in the [HDP/2.0.6/properties/stack_features.json](#) in the branch for that Ambari release. Each feature has a name and description and the stack can provide the minimum and maximum version where that feature is supported.

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

The `stack_tools.json` includes the name and location where the `stack_selector` and `conf_selector` tools are installed.

```
{
  "stack_selector": ["hdp-select", "/usr/bin/hdp-select", "hdp-select"],
  "conf_selector": ["conf-select", "/usr/bin/conf-select", "conf-select"]
}
```

Any custom stack must include these two JSON files. For further information see the [Stack Properties](#) wiki page.

1.13 Services

Each stack-version includes services which are either referenced from *common-services*, or defined inside the stack-version's *services* folder.

Services are defined in *common-services* if they will be shared across multiple stacks. If they will never be shared, then they can be defined inside the stack-version.

1.13.1 Reference *common-services*

To reference a service from *common-services*, the service descriptor file should use the `<extends>` element. (Example: [HDFS in HDP-2.0.6](#))

```

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

```

1.13.2 Define Service

In exactly the same format as services defined in *common-services*, a new service can be defined inside the *services* folder.

Examples:

- [HDFS in BIGTOP-0.8](#)
- [GlusterFS in HDP-2.3.GlusterFS](#)

1.13.3 Extend Service

When a stack-version extends another stack-version, it inherits all details of the parent service. It is also free to override and remove any portion of the inherited service definition.

Examples:

- HDP-2.3 / HDFS - [Adding NFS GATEWAY component, updating service version and OS specific packages](#)
- HDP-2.2 / Storm - [Deleting STORM REST API component, updating service version and OS specific packages](#)
- HDP-2.3 / YARN - [Deleting YARN node-label configuration from capacity-scheduler.xml](#)
- HDP-2.3 / Kafka - [Add Kafka Broker Process alert](#)

1.14 Role Command Order

Role is another name for **Component** (Ex: NAMENODE, DATANODE, RESOURCEMANAGER, HBASE_MASTER, etc.)

As the name implies, it is possible to tell Ambari about the order in which commands should be run for the components defined in your stack.

For example: "ZooKeeper Server should be started before starting NameNode". Or "HBase Master should be started only after NameNode and DataNodes are started".

This can be specified by including the [role command order.json](#) file in the stack-version folder.

1.14.1 Format

Specified in JSON format, the file contains a JSON object with top-level keys being either section names or comments Ex: [HDP-2.0.6](#).

Inside each section object, the key describes the dependent component-action, and the value lists the component-actions which should be done before it.

Structure of role_command_order.json

```
{
  "_comment": "Section 1 comment",
  "section_name_1": {
    "_comment": "Section containing role command orders",
    "<DEPENDENT_COMPONENT_1>-<COMMAND>": ["<DEPENDS_ON_COMPONENT_1>-<COMMAND>",
    "<DEPENDENT_COMPONENT_2>-<COMMAND>": ["<DEPENDS_ON_COMPONENT_3>-<COMMAND>"],
    ...
  },
  "_comment": "Next section comment",
  ...
}
```

1.14.2 Sections

Ambari uses the below sections only:

Section Name	When Used
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)

1.14.3 Commands

Commands currently supported by Ambari are

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

1.14.4 Examples

Role Command Order	Explanation
"HIVE_METASTORE-START": ["MYSQL_SERVER-START", "NAMENODE-START"]	Start MySQL and NameNode components before starting Hive Metastore
"MAPREDUCE_SERVICE_CHECK-SERVICE_CHECK": ["NODEMANAGER-START", "RESOURCEMANAGER-START"],	MapReduce service check needs ResourceManager and NodeManagers started
"ZOOKEEPER_SERVER-STOP" : ["HBASE_MASTER-STOP", "HBASE_REGIONSERVER-STOP", "METRICS_COLLECTOR-STOP"],	Before stopping ZooKeeper servers, make sure HBase Masters, HBase RegionServers and AMS Metrics Collector are stopped.

1.15 Repositories

Each stack-version can provide the location of package repositories to use, by providing a *repos/repoinfo.xml* (Ex: [HDP-2.0.6](#))

The *repoinfo.xml* file contains repositories grouped by operating systems. Each OS specifies a list of repositories that are shown to the user when the stack-version is selected for install.

These repositories are used in conjunction with the [packages defined in a service's metainfo.xml](#) to install appropriate bits on the system.

```
<reposinfo>
  <os family="redhat6">
    <repo>
      <baseurl>http://public-repo-1.hortonworks.com/HDP/centos6/2.x/updates/2.0.
      <repoid>HDP-2.0.6</repoid>
      <reponame>HDP</reponame>
    </repo>
    <repo>
      <baseurl>http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.17/repos/centos6
      <repoid>HDP-UTILS-1.1.0.17</repoid>
      <reponame>HDP-UTILS</reponame>
    </repo>
  </os>
</reposinfo>
```

baseurl- URL of the RPM repository where provided *repoid* can be found

repoid - Repo ID to use that are hosted at *baseurl*

reponame - Display name for the repo being used.

1.15.1.1 Latest Builds

Though repository base URL is capable of providing updates to a particular repo, it has to be defined at build time. This could be an issue later when the repository changes location, or update builds are hosted at a different site.

For such scenarios, a stack-version can provide the location of a JSON file which can provide details of other repo URLs to use.

Example: [HDP-2.3 repoinfo.xml uses <latest> file](#), which then points to alternate repository URLs where latest builds can be found:

```
{
  ...
  "HDP-2.3":{
```

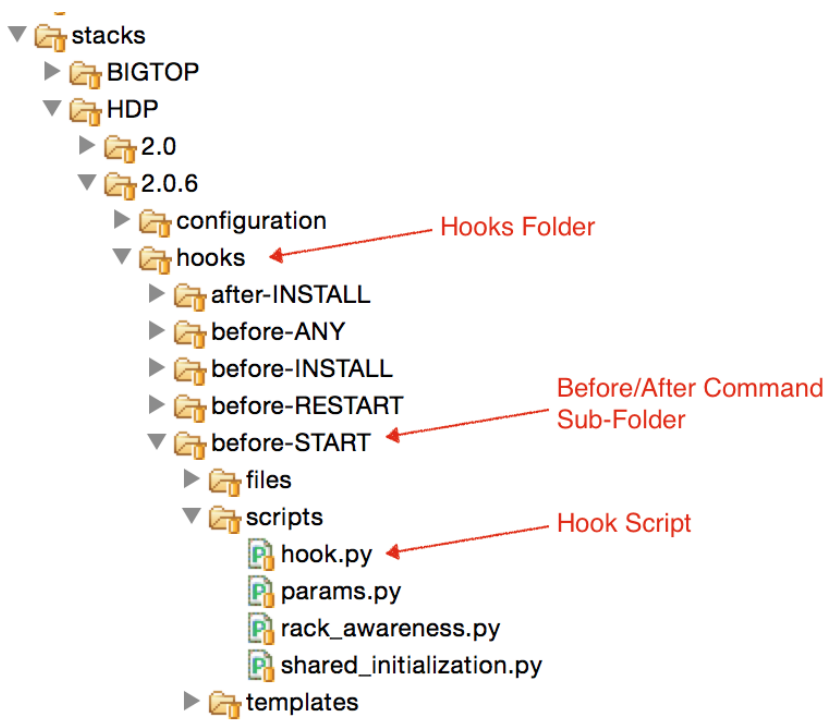
```

"latest":{
    "centos6":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos6/2.x/BUILDS/2.3.6.0-3586/",
    "centos7":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos7/2.x/BUILDS/2.3.6.0-3586/",
    "debian6":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/debian6/2.x/BUILDS/2.3.6.0-3586/",
    "debian7":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/debian7/2.x/BUILDS/2.3.6.0-3586/",
    "suse11":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/suse11sp3/2.x/BUILDS/2.3.6.0-3586/",
    "ubuntu12":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/ubuntu12/2.x/BUILDS/2.3.6.0-3586/",
    "ubuntu14":"http://s3.amazonaws.com/dev.hortonworks.com/HDP/ubuntu14/2.x/BUILDS/2.3.6.0-3586/"
}
},
...
}

```

1.16 Hooks

A stack-version could have very basic and common instructions that need to be run before or after certain Ambari commands, across all services. Instead of duplicating this code across all service scripts and asking users to worry about them, Ambari provides the *Hooks* ability where common before and after code can be pulled away into the *hooks* folder. (Ex: [HDP-2.0.6](#))



1.16.1 Command Sub-Folders

The general naming pattern for *hooks* sub-folders is "<before|after> -<ANY|<CommandName>>". What this means is that the *scripts/hook.py* file under the sub-folder is run either before or after the *command*.

Examples:

Sub-Folder	Purpose	Example
before-START	Hook script called before START command is run on any component of the stack-version.	HDP-2.0.6

Sub-Folder	Purpose	Example
		<ul style="list-style-type: none"> sets up Hadoop log and pid directories creates Java Home symlink Creates <code>/etc/hadoop/conf/topology_script.py</code> etc.
before-INSTALL	Hook script called before installing of any component of the stack-version	HDP-2.0.6 <ul style="list-style-type: none"> Creates repo files in <code>/etc/yum.repos.d</code> Installs basic packages like curl, unzip, etc.

Based on the commands currently supported by Ambari, the following sub-folders can be created based on necessity

Prefix	-	Command	Details
before	-	INSTALL	
		UNINSTALL	
		START	
		RESTART	
		STOP	
after		EXECUTE	
		ABORT	
		UPGRADE	
		SERVICE_CHECK	
		<custom_command>	Custom commands specified by the user like the DECOMMISSION or REBALANCEHDFS commands specified by HDFS

The `scripts/hook.py` script should import [resource_management.libraries.script.hook](#) module and extend the Hook class

```
from resource_management.libraries.script.hook import Hook

class CustomHook(Hook):
    def hook(self, env):
        # Do custom work

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


1.17 Configurations

Though most configurations are set at the service level, there can be configurations which apply across all services to indicate the state of the cluster installed with this stack.

For example, things like ["is security enabled?"](#), ["what user runs smoke tests?"](#) etc.

Such configurations can be defined in the [configuration folder](#) of the stack. They are available for access just like the service-level configs.

1.17.1 Stack Advisor

With each stack containing multiple complex services, it becomes necessary to dynamically determine how the services are laid out on the cluster, and for determining values of certain configurations.

Ambari provides the *Stack Advisor* capability where stacks can write a Python script named *stack-advisor.py* in the *services/* folder. Example: [HDP-2.0.6](#).

Stack-advisor scripts automatically extend the parent stack-version's stack-advisor scripts. This allows newer stack-versions to change behavior without effecting earlier behavior.

Stack advisors provide information on 4 important aspects:

1. Recommend layout of services on cluster
2. Recommend service configurations
3. Validate layout of services on cluster
4. Validate service configurations

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

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

stack-advisor.py Interface

```
1  class StackAdvisor(object):
2      """
3      Abstract class implemented by all stack advisors. Stack advisors advise on stack specific
4  questions.
5
6
7      Currently stack advisors provide following abilities:
8      - Recommend where services should be installed in cluster
9      - Recommend configurations based on host hardware
10     - Validate user selection of where services are installed on cluster
11     - Validate user configuration values
12
13     Each of the above methods is passed in parameters about services and hosts involved as
14 described below.
15
16
17     @type services: dictionary
18     @param services: Dictionary containing all information about services selected by the
19 user.
20
21     Example: {
22         "services": [
23             {
24                 "StackServices": {
```

```

24         "service_name" : "HDFS",
25         "service_version" : "2.6.0.2.2",
26     },
27     "components" : [
28         {
29             "StackServiceComponents" : {
30                 "cardinality" : "1+",
31                 "component_category" : "SLAVE",
32                 "component_name" : "DATANODE",
33                 "display_name" : "DataNode",
34                 "service_name" : "HDFS",
35                 "hostnames" : []
36             },
37             "dependencies" : []
38         }, {
39             "StackServiceComponents" : {
40                 "cardinality" : "1-2",
41                 "component_category" : "MASTER",
42                 "component_name" : "NAMENODE",
43                 "display_name" : "NameNode",
44                 "service_name" : "HDFS",
45                 "hostnames" : []
46             },
47             "dependencies" : []
48         },
49         ...
50     ]
51 },
52 ...
53 ]
54 }
55 @type hosts: dictionary
56 @param hosts: Dictionary containing all information about hosts in this cluster
57 Example: {
58     "items": [
59         {
60             Hosts: {
61                 "host_name": "c6401.ambari.apache.org",
62                 "public_host_name" : "c6401.ambari.apache.org",
63                 "ip": "192.168.1.101",
64                 "cpu_count" : 1,
65                 "disk_info" : [
66                     {
67                         "available" : "4564632",
68                         "used" : "5230344",
69                         "percent" : "54%",
70                         "size" : "10319160",
71                         "type" : "ext4",
72                         "mountpoint" : "/"
73                     },
74                     {

```

```

75         "available" : "1832436",
76         "used" : "0",
77         "percent" : "0%",
78         "size" : "1832436",
79         "type" : "tmpfs",
80         "mountpoint" : "/dev/shm"
81     }
82 ],
83     "host_state" : "HEALTHY",
84     "os_arch" : "x86_64",
85     "os_type" : "centos6",
86     "total_mem" : 3664872
87 }
88 },
89 ...
90 ]
91 }

```

Each of the methods can either return recommendations or validations.

Recommendations are made in a Ambari Blueprints friendly format.

Validations are an array of validation objects.

```

98 """
99
100
101 def recommendComponentLayout(self, services, hosts):
102     """
103     Returns recommendation of which hosts various service components should be installed on.
104
105     This function takes as input all details about services being installed, and hosts
106     they are being installed into, to generate hostname assignments to various components
107     of each service.
108
109
110     @type services: dictionary
111     @param services: Dictionary containing all information about services selected by the
112 user.
113     @type hosts: dictionary
114     @param hosts: Dictionary containing all information about hosts in this cluster
115     @rtype: dictionary
116     @return: Layout recommendation of service components on cluster hosts in Ambari
117 Blueprints friendly format.
118     Example: {
119         "resources" : [
120             {
121                 "hosts" : [
122                     "c6402.ambari.apache.org",
123                     "c6401.ambari.apache.org"
124                 ],
125                 "services" : [

```

```

126         "HDFS"
127     ],
128     "recommendations" : {
129         "blueprint" : {
130             "host_groups" : [
131                 {
132                     "name" : "host-group-2",
133                     "components" : [
134                         { "name" : "JOURNALNODE" },
135                         { "name" : "ZKFC" },
136                         { "name" : "DATANODE" },
137                         { "name" : "SECONDARY_NAMENODE" }
138                     ]
139                 },
140                 {
141                     "name" : "host-group-1",
142                     "components" :
143                         [ { "name" : "HDFS_CLIENT" },
144                           { "name" : "NAMENODE" },
145                           { "name" : "JOURNALNODE" },
146                           { "name" : "ZKFC" },
147                           { "name" : "DATANODE" }
148                         ]
149                 }
150             ]
151         },
152         "blueprint_cluster_binding" : {
153             "host_groups" : [
154                 {
155                     "name" : "host-group-1",
156                     "hosts" : [ { "fqdn" : "c6401.ambari.apache.org" } ]
157                 },
158                 {
159                     "name" : "host-group-2",
160                     "hosts" : [ { "fqdn" : "c6402.ambari.apache.org" } ]
161                 }
162             ]
163         }
164     }
165 }
166 ]
167 }
168 """
169 pass
170
171 def validateComponentLayout(self, services, hosts):
172     """
173     Returns array of Validation issues with service component layout on hosts
174
175     This function takes as input all details about services being installed along with
176

```

```

177         hosts the components are being installed on (hostnames property is populated for
178         each component).
179
180         @type services: dictionary
181         @param services: Dictionary containing information about services and host layout
182         selected by the user.
183         @type hosts: dictionary
184         @param hosts: Dictionary containing all information about hosts in this cluster
185         @rtype: dictionary
186         @return: Dictionary containing array of validation items
187         Example: {
188             "items": [
189                 {
190                     "type" : "host-group",
191                     "level" : "ERROR",
192                     "message" : "NameNode and Secondary NameNode should not be hosted on the same
193 machine",
194                     "component-name" : "NAMENODE",
195                     "host" : "c6401.ambari.apache.org"
196                 },
197                 ...
198             ]
199         }
200         """
201         pass
202
203
204     def recommendConfigurations(self, services, hosts):
205         """
206         Returns recommendation of service configurations based on host-specific layout of
207         components.
208
209         This function takes as input all details about services being installed, and hosts
210         they are being installed into, to recommend host-specific configurations.
211
212
213         @type services: dictionary
214         @param services: Dictionary containing all information about services and component
215         layout selected by the user.
216         @type hosts: dictionary
217         @param hosts: Dictionary containing all information about hosts in this cluster
218         @rtype: dictionary
219         @return: Layout recommendation of service components on cluster hosts in Ambari
220         Blueprints friendly format.
221         Example: {
222             "services": [
223                 "HIVE",
224                 "TEZ",
225                 "YARN"
226             ],
227             "recommendations": {

```

```

228         "blueprint": {
229             "host_groups": [],
230             "configurations": {
231                 "yarn-site": {
232                     "properties": {
233                         "yarn.scheduler.minimum-allocation-mb": "682",
234                         "yarn.scheduler.maximum-allocation-mb": "2048",
235                         "yarn.nodemanager.resource.memory-mb": "2048"
236                     }
237                 },
238                 "tez-site": {
239                     "properties": {
240                         "tez.am.java.opts": "-server -Xmx546m -Djava.net.preferIPv4Stack=true -
241XX:+UseNUMA -XX:+UseParallelGC",
242                         "tez.am.resource.memory.mb": "682"
243                     }
244                 },
245                 "hive-site": {
246                     "properties": {
247                         "hive.tez.container.size": "682",
248                         "hive.tez.java.opts": "-server -Xmx546m -Djava.net.preferIPv4Stack=true -
249XX:NewRatio=8 -XX:+UseNUMA -XX:+UseParallelGC",
250                         "hive.auto.convert.join.noconditionaltask.size": "238026752"
251                     }
252                 }
253             },
254             "blueprint_cluster_binding": {
255                 "host_groups": []
256             }
257         },
258         "hosts": [
259             "c6401.ambari.apache.org",
260             "c6402.ambari.apache.org",
261             "c6403.ambari.apache.org"
262         ]
263     }
264 }
265 ""
266 pass
267
268 def validateConfigurations(self, services, hosts):
269     """
270     Returns array of Validation issues with configurations provided by user
271     This function takes as input all details about services being installed along with
272     configuration values entered by the user. These configurations can be validated against
273     service requirements, or host hardware to generate validation issues.
274
275
276     @type services: dictionary
277     @param services: Dictionary containing information about services and user
278 configurations.

```

```

279     @type hosts: dictionary
280     @param hosts: Dictionary containing all information about hosts in this cluster
281     @rtype: dictionary
282     @return: Dictionary containing array of validation items
           Example: {
               "items": [
                   {
                       "config-type": "yarn-site",
                       "message": "Value is less than the recommended default of 682",
                       "type": "configuration",
                       "config-name": "yarn.scheduler.minimum-allocation-mb",
                       "level": "WARN"
                   }
               ]
           }
           """"
           pass

```

1.17.1.1 Examples

- [Stack Advisor interface](#)
- [Default Stack Advisor implementation - for all stacks](#)
- [HDP \(2.0.6\) Default Stack Advisor implementation](#)
- [YARN container size calculated](#)
- Recommended configurations - [HDFS](#), [YARN](#), [MapReduce2](#), [HBase](#) (HDP-2.0.6), [HBase](#) (HDP-2.3)
- [Delete HBase Bucket Cache configs on smaller machines](#)
- [Specify maximum value for Tez config](#)

1.18 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. Most of these properties were introduced in Ambari 2.4 version in the effort of parameterize stack information and facilitate the reuse of common-services code by other distributions.

Such properties can be defined in .json format in the [properties folder](#) of the stack.

More details about stack properties can be found on [Stack Properties section](#).

1.19 Widgets

At the stack-version level one can contribute heatmap entries to the main dashboard of the cluster. Generally these heatmaps would be ones which apply to all services - like host level heatmaps.

Example: [HDP-2.0.6 contributes host level heatmaps](#)

1.20 Kerberos

We have seen previously the Kerberos descriptor at the service level.

One can be defined at the stack-version level also to describe identities across all services.

Read more about the Kerberos support and the Kerberos Descriptor in the [Automated Kerberization](#) page.

Example: [Smoke tests user and SPNEGO user defined in HDP-2.0.6](#)

1.21 Stack Upgrades

Ambari provides the ability to upgrade your cluster from a lower stack-version to a higher stack-version. Each stack-version can define *upgrade-packs*, which are XML files describing the upgrade process. These *upgrade-pack* XML files are placed in the stack-version's *upgrades/* folder.

Example: [HDP-2.3](#), [HDP-2.4](#)

Each stack-version should have an upgrade-pack for the next stack-version a cluster can **upgrade to**.
Ex: [Upgrade-pack from HDP-2.3 to HDP-2.4](#)

There are two types of upgrades:

Upgrade Type	Pros	Cons
Rolling Upgrade (RU)	Minimal cluster downtime – services available throughout upgrade process	Takes time (sometimes days depending on cluster size) due to incremental upgrade approach
Express Upgrade (EU)	Cluster unavailable – services are stopped during upgrade process	Much faster – clusters can be upgraded in a couple of hours

Each component which has to be upgraded by Ambari should specify the **versionAdvertised** flag in the *metainfo.xml*.

This will tell Ambari to use the component's version and perform upgrade. Not specifying this flag will result in Ambari not upgrading the component.

Example: [HDFS NameNode](#) (versionAdvertised=true), [AMS Metrics Collector](#) (versionAdvertised=false).

1.21.1 Rolling Upgrades

In Rolling Upgrade each service is upgraded with minimal downtime in mind. The general approach is to quickly upgrade the master components, followed by upgrading of workers in batches. The service will not be available when masters are restarting. However when master components are in High Availability (HA), the service continues to be available through restart of each master.

You can read more about the Rolling Upgrade process via this [blog post](#) and [documentation](#).

Examples

- [HDP-2.2.x to HDP-2.2.y](#)
- [HDP-2.2 to HDP-2.3](#)
- [HDP-2.2 to HDP-2.4](#)
- [HDP-2.3 to HDP-2.4](#)

1.21.2 Express Upgrades

In Express Upgrade the goal is to upgrade entire cluster as fast as possible – even if it means cluster downtime. It is generally much faster than Rolling Upgrade.

For each service the components are first stopped, upgraded and then started.

You can read about Express Upgrade steps in this [documentation](#).

Examples

- [HDP-2.1 to HDP-2.3](#)
- [HDP-2.2 to HDP-2.4](#)

What is a config-type?

String representing a group of configurations. Example: *core-site*, *hdfs-site*, *yarn-site*, etc. When configurations are saved in Ambari, they are persisted within a version of config-type which is immutable. If you change and save HDFS core-site configs 4 times, you will have 4 versions of config-type core-site. Also, when a service's configs are saved, only the changed config-types are updated. The supported types of config files are xml, env and properties.

1.1 Adding / modifying config properties in a config-type.

There are a number of supported property and value attributes in the XML schema of an Ambari managed config type. These attributes can be useful in specifying type , constraints, upgrade related choices, validations etc. A config property in a config-type looks like this.

```
<property require-input="false">
  <name>dfs.namenode.checkpoint.dir</name>
  <value>/hadoop/hdfs/namesecondary</value>
  <description>Determines where on the local filesystem the DFS secondary
    name node should store the temporary images to merge.
    If this is a comma-delimited list of directories then the image is
    replicated in all of the directories for redundancy.
</description>
  <display-name>SecondaryNameNode Checkpoint directories</display-name>
  <filename>hdfs-site.xml</filename>
  <deleted>false</deleted>
  <on-ambari-upgrade add="false" delete="false" update="false"/>
  <on-stack-upgrade merge="true"/>
  <property-type></property-type>
  <value-attributes>
    <type>directories</type>
    <overridable>false</overridable>
    <keystore>false</keystore>
  </value-attributes>
  <depends-on/>
  <property_depended_by/>
  <used-by/>
</property>
```

The configuration schema XSD can be found here - <https://github.com/apache/ambari/blob/trunk/ambari-server/src/main/resources/configuration-schema.xsd>

The corresponding Java class that maps these property definitions is <https://github.com/apache/ambari/blob/trunk/ambari-server/src/main/java/org/apache/ambari/server/state/PropertyInfo.java>. The following table discusses the supported basic and advanced attributes.

Property Key	Explanation	Mandatory (M) or Optional (O)	Sample / Expected values
name	Name of the property	M	dfs.namenode.checkpoint.dir
value	Value of the property	M	/hadoop/hdfs/secondary
description	A short description of the property.	M	Determines where on the local filesystem the DFS secondary name node should store the temporary images to merge.
display-name	Display name of the property as seen on Ambari UI.	O	SecondaryNameNode Checkpoint directories
filename	Name of the file on disk to which the config-type is serialized	O	
deleted	Deprecated property.	O	
require-input	Does the property mandate a value or an empty value is allowed?	O	true / false. Default = false.
on-ambari-upgrade	Do you want to add it to the service config	O	add="false" delete="false" update="false".

Property Key	Explanation	Mandatory (M) or Optional (O)	Sample / Expected values
	<p>when an Ambari upgrade is done?</p> <p>Note : A service config change will trigger a Restart required for the service on the Ambari UI.</p>		
on-stack-upgrade	Do you want to ambari to add it to the service config when a stack upgrade is done?	O	merge="true"
property-type	Type of property.	O	<p>PASSWORD, USER, UID, GROUP, GID, TEXT, ADDITIONAL_USER_PROPERTY, NOT_MANAGED_HDFS_PATH, VALUE_FROM_PROPERTY_FILE, KERBEROS_PRINCIPAL.</p> <p>Default = TEXT</p>
value-attributes	Set of attributes for the value of the property.	O	<explained below>

Property Key	Explanation	Mandatory (M) or Optional (O)	Sample / Expected values
depends-on	Used to specify dependent properties, even across config types. This is used in stack recommendations when the user changes a property, and another property needs to be updated accordingly.	O	<pre> <depends-on> <property> <type>zoo.cfg</type> <name>clientPort</name> </property> </depends-on> </pre>
property_dependent_by	The reverse dependency mapping. Some other property is dependent on this property.	O	<pre> <property_dependent_by> <dependentByProperties> <name>hive.exec.orc.encoding.strategy</name> <type>hive-site</type> </dependentByProperties> </property_dependent_by> </pre>
used-by		O	

The set of supported value-attributes in Ambari are explained below.

Some of these are used in enhanced configs - [Enhanced Configs](#).

Value Attribute	Explanation	Sample / Allowed value
type	Type of the value.	boolean / int / float / directory / directories / content / value-list / user / password
overridable		true / false

Value Attribute	Explanation	Sample / Allowed value
empty_value_valid	Is an empty value is valid?	true / false
ui_only_property		true / false
read_only	Uneditable	true / false
editable_only_at_install	Value can be edited only during the initial cluster install. For example, HBase table split points.	true / false
show_property_name		true / false
increment_step		
selection_cardinality		
property-file-name		
property-file-type		
entries		
hidden		
entries_editable		true / false
user-groups		
keystore	Keystore enabled or not.	true / false
maximum	Max allowed value	
minimum	Min allowed value	
unit	Unit of the value	B / MB / ms / Bytes / milliseconds /
visible		
copy		

1.2 Background

At present, stack definitions are bundled with Ambari core and are part of Apache Ambari releases. This enforces having to do an Ambari release with updated stack definitions whenever a new version of a stack is released. Also to add an "add-on" service (custom service) to the stack definition, one has to manually add the add-on service to the stack definition on an Ambari Server. There is no release vehicle that can be used to ship add-on services.

Apache Ambari Management Packs addresses this issue by decoupling Ambari's core functionality (cluster management and monitoring) from stack management and definition. An Apache Ambari Management Pack (Mpack) can bundle multiple service definitions, stack definitions, stack add-on service definitions, view definitions services so that releasing these artifacts don't enforce an Apache Ambari release. Apache Ambari Management Packs will be released as separate release artifacts and will follow its own release cadence instead of being tightly coupled with Apache Ambari releases.

Management packs are released as tarballs, however they contain a metadata file (mpack.json) that specify the contents of the management pack and actions to perform when installing the management pack.

1.3 Apache JIRA

[AMBARI-14854](#)

1.4 Release Timelines

- Short Term Goals (Apache Ambari 2.4.0.0 release)
 1. Provide a release vehicle for stack definitions (example: HDP management pack, IOP management pack).
 2. Provide a release vehicle for add-on/custom services (example: Microsoft-R management pack)
 3. Retrofit in existing stack processing infrastructure
 4. Command line to update stack definitions and service definitions
- Long Term Goals (Ambari 2.4+)
 1. Release HDP stacks as mpacks
 2. Build management pack processing infrastructure that will replace the stack processing infrastructure.
 3. Dynamic creation of stack definitions by processing management packs
 4. Provide a REST API adding/removing /upgrading management packs.

1.5 Management Pack Metadata (Mpack.json)

Management pack should contain following metadata information in mpack.json.

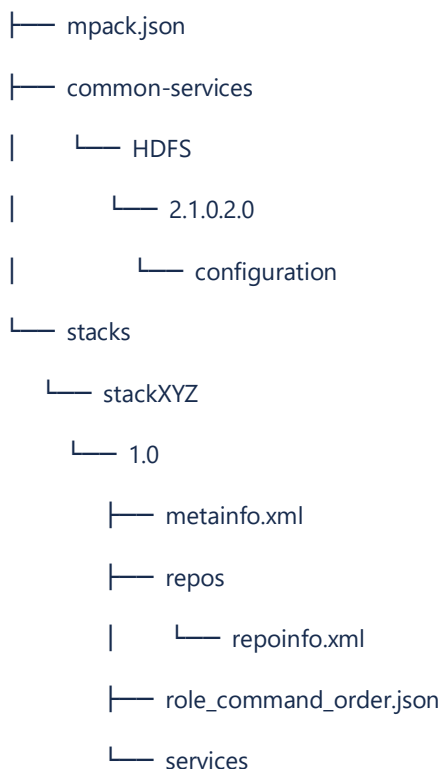
- **Name:** Unique management pack name
- **Version:** Management pack version
- **Description:** Friendly description of the management pack
- **Prerequisites:**
 - Minimum Ambari Version on which the management pack is installable.
 - **Example:** To install stackXYZ-ambari-mpack1.0.0.0 management pack, Ambari should be atleast on Apache Ambari 2.4.0.0 version.
 - Minimum management pack version that should be installed before upgrading to this management pack.

- **Example:** To upgrade to stackXYZ-ambari-mpack-2.0.0.0 management pack, stackXYZ--ambari-mpack-1.8.0.0 management pack or higher should be installed.
- Minimum stack version that should already be present in the stack definitions for this management pack to be applicable.
- **Example:** To add a add-on service management pack myservice-ambari-mpack-1.0.0.0 management pack stackXYZ-2.1 stack definition should be present.
- **Artifacts:**
 - List of release artifacts (service definitions, stack definitions, stack-addon-service-definitions, view-definitions) bundled in the management pack.
 - Metadata for each artifact like source directory, additional applicability for that artifact etc.
 - Supported Artifact Types
 - **service-definitions:** Contains service definitions similar to common-services/serviceA/1.0.0
 - **stack-definitions:** Contains stack definitions similar to stacks/stackXYZ/1.0
 - **extension-definitions:** Contains dynamic stack extensions (Refer: [Extensions](#))
 - **stack-addon-service-definitions:** Defines add-on service applicability for stacks and how to merge the add-on service into the stack definition.
 - **view-definitions** (Not supported in Apache Ambari 2.4.0.0)
 - A management pack can have more than one release artifacts.
 - **Example:** It should be possible to create a management pack that bundles together
 - **stack-definitions:** stackXYZ-1.0, stackXYZ-1.1, stackXYZ-2.0
 - **service-definitions:** HAWQ, HDFS, ZOOKEEPER
 - **stack-addon-service-definitions:** HAWQ/2.0.0 is applicable to stackXYZ-2.0, stackABC-1.0
 - **view-definitions:** Hive, Jobs, Slider (Apache Ambari 2.4.0.0)

1.6 Management Pack Structure

1.6.1 StackXYZ Management Pack Structure

stackXYZ-ambari-mpack-1.0.0.0



```

├── HDFS
|   ├── configuration
|   |   └── hdfs-site.xml
|   └── metainfo.xml
├── stack_advisor.py
└── ZOOKEEPER
    └── metainfo.xml

```

1.6.2 StackXYZ Management Pack Mpack.json

stackXYZ-ambari-mpack1.0.0.0/mpack.json

```

{
  "type" : "full-release",
  "name" : "stackXYZ-ambari-mpack",
  "version": "1.0.0.0",
  "description" : "StackXYZ Management Pack",
  "prerequisites": {
    "min_ambari_version" : "2.4.0.0"
  },
  "artifacts": [
    {
      "name" : "stackXYZ-service-definitions",
      "type" : "service-definitions",
      "source_dir": "common-services"
    },
    {
      "name" : "stackXYZ-stack-definitions",
      "type" : "stack-definitions",
      "source_dir": "stacks"
    }
  ]
}

```

1.6.3 Add-On Service Management Pack Structure

myservice-ambari-mpack-1.0.0.0

```

├── common-services

```



```

|   └─ MYSERVICE
|       └─ 1.0.0
|           └─ configuration
|               └─ myserviceconfig.xml
|           └─ metainfo.xml
|           └─ package
|               └─ scripts
|                   └─ client.py
|                   └─ master.py
|                   └─ slave.py
|           └─ role_command_order.json
└─ custom-services
    └─ MYSERVICE
        └─ 1.0.0
            └─ metainfo.xml
        └─ 2.0.0
            └─ metainfo.xml
└─ mpack.json

```

1.6.4 Add-On Service Management Pack Mpack.json

myservice-ambari-mpack-1.0.0.0/mpack.json

```

{
  "type" : "full-release",
  "name" : "myservice-ambari-mpack",
  "version": "1.0.0.0",
  "description" : "MyService Management Pack",
  "prerequisites": {
    "min-ambari-version" : "2.4.0.0",
    "min-stack-versions" : [
      {
        "stack_name" : "stackXYZ",
        "stack_version" : "2.2"
      }
    ]
  }
}

```

```
},
"artifacts": [
  {
    "name" : "MYSERVICE-service-definition",
    "type" : "service-definition",
    "source_dir" : "common-services/MYSERVICE/1.0.0",
    "service_name" : "MYSERVICE",
    "service_version" : "1.0.0"
  },
  {
    "name" : "MYSERVICE-1.0.0",
    "type" : "stack-addon-service-definition",
    "source_dir": "addon-services/MYSERVICE/1.0.0",
    "service_name" : "MYSERVICE",
    "service_version" : "1.0.0",
    "applicable_stacks" : [
      {
        "stack_name" : "stackXYZ", "stack_version" : "2.2"
      }
    ]
  },
  {
    "name" : "MYSERVICE-2.0.0",
    "type" : "stack-addon-service-definition",
    "source_dir": "custom-services/MYSERVICE/2.0.0",
    "service_name" : "MYSERVICE",
    "service_version" : "2.0.0",
    "applicable_stacks" : [
      {
        "stack_name" : "stackXYZ",
        "stack_version" : "2.4"
      }
    ]
  }
]
}
```

1.7 Installing Management Pack

```
ambari-server install-mpack --mpack=/path/to/mpack.tar.gz --purge --verbose
```

Note: Do not pass the "--purge" command line parameter when installing an add-on service management pack. The "--purge" flag is used to purge any existing stack definition (HDP is included in Ambari release) and should be included only when installing a Stack Management Pack.

1.8 Upgrading Management Pack

```
ambari-server upgrade-mpack --mpack=/path/to/mpack.tar.gz --verbose
```

Each stack version must provide a *metainfo.xml* descriptor file which can declare whether the stack inherits from another stack version:

```
<metainfo>
  <versions>
    <active>true</active>
  </versions>
  <extends>2.3</extends>
  ...
</metainfo>
```

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

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 stack version are merged with the parent stack 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 stack version replace those from the parent stack 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 should be defined in every stack version. The upgrades directory and its corresponding XML files should be defined in all stack versions that support upgrade.

1.9 Services Folder

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)
        ...
```

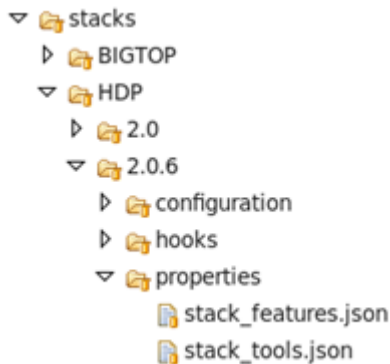
Services defined within the services folder follow the rules for [service inheritance](#). By default if a service does not declare an explicit inheritance (via the **extends** tag), the service will inherit from the service defined at the parent stack version.

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. Most of these properties were introduced in Ambari 2.4 version in the effort of parameterize stack information and facilitate the reuse of common-services code by other distributions.

Such properties can be defined in .json format in the [properties folder](#) of the stack.



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, Phoenix)...

Stack featurization was added as part of the HDP stack configurations on [HDP/2.0.6/configuration/cluster-env.xml](#), introducing a new stack_features property which value is processed in the stack engine from an external property file.

/HDP/2.0.6/configuration/cluster-env.xml

```
<!-- Define stack_features property in the base stack. DO NOT override this property for each stack
version -->
<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 are defined in [stack_features.json](#) file under /HDP/2.0.6/properties. These features support is now available for access at service-level code to change certain service behaviors or configurations. This is an example of features described in stack_features.json file:

stack_features.json

```
"stack_features": [  
  {  
    "name": "snappy",  
    "description": "Snappy compressor/decompressor support",  
    "min_version": "2.0.0.0",  
    "max_version": "2.2.0.0"  
  },  
  {  
    "name": "lzo",  
    "description": "LZO libraries support",  
    "min_version": "2.2.1.0"  
  },  
  {  
    "name": "express_upgrade",  
    "description": "Express upgrade support",  
    "min_version": "2.1.0.0"  
  },  
  {  
    "name": "rolling_upgrade",  
    "description": "Rolling upgrade support",  
    "min_version": "2.2.0.0"  
  }  
]
```

where min_version/max_version are optional constraints.

Feature constants, matching features names, such as ROLLING_UPGRADE = "rolling_upgrade" has been added to a new StackFeature class in [resource management/libraries/functions/constants.py](#)

class StackFeature

```
class StackFeature:  
    """  
    Stack Feature supported  
    """  
    SNAPPY = "snappy"  
    LZ0 = "lzo"  
    EXPRESS_UPGRADE = "express_upgrade"  
    ROLLING_UPGRADE = "rolling_upgrade"
```

Additionally, corresponding helper functions has been introduced in [resource management/libraries/functions/stack_fetaures.py](#) to parse the json file content and called from service code to check if the stack supports the specific feature.

This is an example where the new stack featurization design is used in service code:

stack featurization example

```
if params.version and check_stack_feature(StackFeature.ROLLING_UPGRADE, params.v
    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 under /HDP/2.0.6/properties

And declared as part of the HDP stack configurations as a new property on [/HDP/2.0.6/configuration/cluster-env.xml](#)

/HDP/2.0.6/configuration/cluster-env.xml

```
<!-- Define stack_tools property in the base stack. DO NOT override this property for each stack
version -->
<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 [resource_management/libraries/functions/stack_tools.py](#). These helper functions are used to remove hardcodings in resource_management library.

Structure

metainfo.xml is a declarative definition of an Ambari managed service describing its content. It is the most critical file for any service definition. This section describes various key sub-sections within a metainfo.xml file.

Non-mandatory fields are described in italics.

The top level fields to describe a service are as follows:

Field	What is it used for	Sample Values
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.	HDFS
displayName	the display name of the service	HDFS
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.	2.1.0.2.0
components	the list of component that the service is comprised of	<check out HDFS metainfo>
osSpecifics	OS specific package information for the service	<check out HDFS metainfo>
<i>commandScript</i>	service level commands may also be defined. The command is executed on a component instance that is a client	<check out HDFS metainfo>
<i>comment</i>	a short description describing the service	Apache Hadoop Distributed File System
<i>requiredServices</i>	what other services that should be present on the cluster	<check out HDFS metainfo>
<i>configuration-dependencies</i>	configuration files that are expected by the service (config files owned by other services are specified in this list)	<check out HDFS metainfo>
<i>restartRequiredAfterRackChange</i>	Restart Required After Rack Change	true / false
<i>configuration-dir</i>	Use this to specify a different config directory if not 'configuration'	-

1.1

service/components

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

Field	What is it used for	Sample Values
name	name of the component	HDFS
displayName	display name of the component.	HDFS
category	type of the component - MASTER, SLAVE, and CLIENT	2.1.0.2.0
commandScript	application wide commands may also be defined. The command is executed on a component instance that is a client	<check out HDFS metainfo>

<i>cardinality</i>	allowed/expected number of instances	For example, 1-2 for MASTER, 1+ for Slave
<i>reassignAllowed</i>	whether the component can be reassigned / moved to a different host.	true / false
<i>versionAdvertised</i>	does the component advertise its version - used during rolling/express upgrade	Apache Hadoop Distributed File System
<i>timelineAppid</i>	This will be the component name under which the metrics from this component will be collected.	<check out HDFS metainfo>
<i>dependencies</i>	the list of components that this component depends on	<check out HDFS metainfo>
<i>customCommands</i>	a set of custom commands associated with the component in addition to standard commands.	RESTART_LLAP (Check out HIVE metainfo)

1.2 service/osSpecifics

- OS specific package names (rpm or deb packages)

Field	What is it used for	Sample Values
osFamily	the os family for which the package is applicable	any => all amazon2015, redhat6, debian7, ubuntu12, ubuntu14, ubuntu16
packages	list of packages that are needed to deploy the service	<check out HDFS metainfo>
package/name	name of the package (will be used by the yum/zypper/apt commands)	Eg hadoop-lzo.

1.3 Service/commandScript –

the script that implements service check

Field	What is it used for	Sample Values
<i>script</i>	the relative path to the script	<pre> <commandScript> <script>scripts/service_check.py</script> <scriptType>PYTHON</scriptType> <timeout>300</timeout> </commandScript> </pre>
<i>scriptType</i>	the type of the script, currently only supported type is PYTHON	
<i>timeout</i>	custom timeout for the command - this supersedes ambari default	

1.4 Dependency

service/component/dependencies/dependency

Field	What is it used for	Sample Values
<i>name</i>	name of the component it depends on	<pre> <dependency> <name>HDFS/ZKFC</name> <scope>cluster</scope> <auto-deploy> <enabled>>false</enabled> </auto-deploy> <conditions> </pre>
<i>scope</i>	cluster / host. specifies whether the dependent component should be present in the same cluster or the same host.	

auto-deploy	custom timeout for the command - this supersedes ambari default	<pre> <condition xsi:type="propertyExists"> <configType>hdfs-site</configType> <property>dfs.nameservices</property> </condition> </conditions> </dependency> </pre>
conditions	Conditions in which this dependency exists. For example, the presence of a property in a config.	

1.5 service/component/commandScript

- the script that implements components specific default commands (Similar to service/commandScript)

1.6 service/component/logs

- provides log search integration.

logId	logid of the component	<pre> <log> <logId>hdfs_namenode</logId> <primary>true</primary> </log> </pre>
primary	primary log id or not.	

1.7 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

1.8 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)

FAQ

- [\[STACK\]/\[SERVICE\]/metainfo.xml](#)
- [If a component exists in the parent stack and is defined again in the child stack with just a few attributes, are these values just to override the parent's values or the whole component definition is replaced?](#)
[What about other elements in metainfo.xml -- which rules apply?](#)
- [If a component is missing in the new definition but is present in the parent, does it get inherited?](#)
- [Configuration dependencies for the service -- are they overwritten or merged?](#)

1.1 [STACK]/[SERVICE]/metainfo.xml

Sample metainfo.xml

```
<metainfo>
<schemaVersion>2.0</schemaVersion>
<services>
  <service>
    <name>HBASE</name>
    <displayName>HBase</displayName>
    <comment>Non-relational distributed database and centralized service for
configuration management & synchronization
    </comment>
    <version>0.96.0.2.0</version>
    <components>
      <component>
        <name>HBASE_MASTER</name>
        <displayName>HBase Master</displayName>
        <category>MASTER</category>
        <cardinality>1+</cardinality>
        <versionAdvertised>true</versionAdvertised>
        <timelineAppid>HBASE</timelineAppid>
        <dependencies>
          <dependency>
            <name>HDFS/HDFS_CLIENT</name>
            <scope>host</scope>
            <auto-deploy>
              <enabled>true</enabled>
            </auto-deploy>
          </dependency>
          <dependency>
            <name>ZOOKEEPER/ZOOKEEPER_SERVER</name>
            <scope>cluster</scope>
            <auto-deploy>
              <enabled>true</enabled>
              <co-locate>HBASE/HBASE_MASTER</co-locate>
            </auto-deploy>
          </dependency>
        </dependencies>
        <commandScript>
          <script>scripts/hbase_master.py</script>
          <scriptType>PYTHON</scriptType>
          <timeout>1200</timeout>
        </commandScript>
        <customCommands>
          <customCommand>
            <name>DECOMMISSION</name>
            <commandScript>
              <script>scripts/hbase_master.py</script>
              <scriptType>PYTHON</scriptType>
            </commandScript>
          </customCommand>
        </customCommands>
      </component>
    </components>
  </service>
</services>
</metainfo>
```

```

        <timeout>600</timeout>
      </commandScript>
    </customCommand>
  </customCommands>
</component>

<component>
  <name>HBASE_REGIONSERVER</name>
  <displayName>RegionServer</displayName>
  <category>SLAVE</category>
  <cardinality>1+</cardinality>
  <versionAdvertised>true</versionAdvertised>
  <timelineAppid>HBASE</timelineAppid>
  <commandScript>
    <script>scripts/hbase_regionserver.py</script>
    <scriptType>PYTHON</scriptType>
  </commandScript>
</component>

<component>
  <name>HBASE_CLIENT</name>
  <displayName>HBase Client</displayName>
  <category>CLIENT</category>
  <cardinality>1+</cardinality>
  <versionAdvertised>true</versionAdvertised>
  <commandScript>
    <script>scripts/hbase_client.py</script>
    <scriptType>PYTHON</scriptType>
  </commandScript>
  <configFiles>
    <configFile>
      <type>xml</type>
      <fileName>hbase-site.xml</fileName>
      <dictionaryName>hbase-site</dictionaryName>
    </configFile>
    <configFile>
      <type>env</type>
      <fileName>hbase-env.sh</fileName>
      <dictionaryName>hbase-env</dictionaryName>
    </configFile>
  </configFiles>
</component>
</components>

<osSpecifics>
  <osSpecific>
    <osFamily>any</osFamily>
    <packages>
      <package>
        <name>hbase</name>
      </package>
    </packages>
  </osSpecific>
</osSpecifics>

<commandScript>
  <script>scripts/service_check.py</script>
  <scriptType>PYTHON</scriptType>
  <timeout>300</timeout>
</commandScript>

<requiredServices>
  <service>ZOOKEEPER</service>
  <service>HDFS</service>
</requiredServices>

```

```
<configuration-dependencies>
  <config-type>core-site</config-type>
  <config-type>hbase-site</config-type>
  <config-type>ranger-hbase-policymgr-ssl</config-type>
  <config-type>ranger-hbase-security</config-type>
</configuration-dependencies>

</service>
</services>
</metainfo>
```

1.2 Override rule

If a component exists in the parent stack and is defined again in the child stack with just a few attributes, are these values just to override the parent's values or the whole component definition is replaced? What about other elements in metainfo.xml -- which rules apply?

Ambari goes property by property and merge them from parent to child. So if you remove a category for example from the child it will be inherited from parent, that goes for pretty much all properties. So, the question is how do we tackle existence of a property in both parent and child. Here, most of the decision still follow same paradigm as take the child value instead of parent and every property in parent, not explicitly deleted from child using a marker like `<deleted>` tag, is included in the merge.

- For config-dependencies, we take all or nothing approach, if this property exists in child use it and all of its children else take it from parent.
- The custom commands are merged based on names, such that merged definition is a union of commands with child commands with same name overriding those from parent.
- Cardinality is overwritten by a child or take from the parent if child has not provided one.

You could look at this method for more details: `org.apache.ambari.server.api.util.StackExtensionHelper#mergeServices`

For more information see the [Service Inheritance](#) wiki page.

1.3 Overlapping rule

If a component is missing in the new definition but is present in the parent, does it get inherited?

Generally yes.

1.4 Overwritten rule

Configuration dependencies for the service -- are they overwritten or merged?

Overwritten.

HOOK

A stack can add during the following points in Ambari actions:

- before ANY
- before and after INSTALL
- before RESTART
- before START

As mentioned in [Stack Inheritance](#), the hooks directory if defined in the current stack version replace those from the parent stack version. 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 hooks directory should have 5 sub-directories:

- after-INSTALL
- before-ANY
- before-INSTALL
- before-RESTART
- before-START

Each hook directory can have 3 sub-directories:

- files
- scripts
- templates

The scripts directory is the main directory and must be supplied. This must contain a hook.py file. It may contain other python scripts which initialize variables (like a params.py file) or other utility files contain functions used in the hook.py file.

The files directory can any non-python files such as scripts, jar or properties files.

The templates folder can contain any required j2 files that are used to set up properties files.

The hook.py file should extend the Hook class defined in `resource_management/libraries/script/hook.py`. The naming convention is to name the hook class after the hook folder such as `AfterInstallHook` if the class is in the after-INSTALL folder. The hook.py file must define the `hook(self, env)` function. Here is an example hook:

```
from resource_management.libraries.script.hook import Hook

class AfterInstallHook(Hook):

    def hook(self, env):

        import params

        env.set_params(params)

        # Call any functions to set up the stack after install

if __name__ == "__main__":

    AfterInstallHook().execute()
```

Upgrade

Especially during upgrade, it is important to be able to set the current stack and configuration versions. For non-custom services, these functions are implemented in the `conf-select` and `stack-select` functions. These can be imported in a service's scripts with the following imports:

```
from resource_management.libraries.functions import conf_select
from resource_management.libraries.functions import stack_select
```

Typically the select functions, which is used to set the stack and configuration versions, are called in the `pre_upgrade_restart` function during a rolling upgrade:

```
def pre_upgrade_restart(self, env, upgrade_type=None):
    import params
    env.set_params(params)

    # this function should not execute if the version can't be determined or
    # the stack does not support rolling upgrade
    if not (params.version and check_stack_feature(StackFeature.ROLLING_UPGRADE, params.version)):
        return

    Logger.info("Executing <My Service> Stack Upgrade pre-restart")
    conf_select.select(params.stack_name, "<my_service>", params.version)
    stack_select.select("<my_service>", params.version)
```

The select functions will set up symlinks for the current stack or configuration version. For the stack, this will set up the links from the stack root current directory to the particular stack version. For example:

```
/usr/hdp/current/hadoop-client -> /usr/hdp/2.5.0.0/hadoop
```

For the configuration version, this will set up the links for all the configuration directories, as follows:

```
/etc/hadoop/conf -> /usr/hdp/current/hadoop-client/conf
/usr/hdp/current/hadoop-client/conf -> /etc/hadoop/2.5.0.0/0
```

The `stack_select` and `conf_select` functions can also be used to return the hadoop directories:

```
hadoop_prefix = stack_select.get_hadoop_dir("home")
hadoop_bin_dir = stack_select.get_hadoop_dir("bin")
hadoop_conf_dir = conf_select.get_hadoop_conf_dir()
```

The `conf_select` api is as follows:

```
def select(stack_name, package, version, try_create=True, ignore_errors=False)
def get_hadoop_conf_dir(force_latest_on_upgrade=False)
```

The `stack_select` api is as follows:


```
def select(component, version)
```

```
def get_hadoop_dir(target, force_latest_on_upgrade=False)
```

Unfortunately for custom services these functions are not available to set up their configuration or stack versions. A custom service could implement their own functions to set up the proper links.