Apache Karaf Cellar 4.x - Documentation

Apache Software Foundation

Apache Karaf Cellar 4.x - Documentation

User Guide

Architecture Guide

User Guide

1. Introduction

1.1. Use Cases

The first goal of Karaf Cellar is to synchronize the status of several Karaf instances (named nodes).

Cellar provides dedicated shell commands and JMX MBeans to manage the cluster, and manipulate the resources on the cluster.

It’s also possible to enable local resources listeners: these listeners broadcast local resource changes as cluster events. Please note that this behavior is disabled by default as it can have side effects (especially when a node is stopped). Enabling listeners is at your own risk.

The nodes list could be discovered (using unicast or multicast), or "static" defined (using a couple hostname or IP and port list).

Cellar is able to synchronize: * bundles (remote or local) * config * features

Optionally, Cellar also support synchronization of OSGi EventAdmin, OBR (URLs and bundles).

The second goal is to provide a Distributed OSGi runtime. It means that using Cellar, you are able to call an OSGi service located on a remote instance. See the transport and DOSGi section of the user guide.

Finally, Cellar also provides "runtime clustering" by providing dedicated feature like: * HTTP load balancing * HTTP sessions replication * log centralization Please, see the sections dedicated to those features.

1.2. Cross topology

This is the default Cellar topology. Cellar is installed on all nodes, each node has the same function.

It means that you can perform actions on any node, it will be broadcasted to all others nodes.

1.3. Star topology

In this topology, if Cellar is installed on all nodes, you perform actions only on one specific node (the "manager").

To do that, the "manager" is a standard Cellar node, and the event producing is disable on all others nodes (`cluster:producer-stop` on all "managed" nodes).

Like this, only the "manager" will send event to the nodes (which are able to consumer and handle), but no event can be produced on the nodes.

2. Installation

This chapter describes how to install Apache Karaf Cellar into your existing Karaf based installation.

2.1. Pre-Installation Requirements

Cellar is installed on running Karaf instances.

Cellar is provided as a Karaf features descriptor. The easiest way to install is just to have an internet connection from the Karaf running instance.

See link:deploy to how to install and start Cellar.

2.2. Building from Sources

If you intend to build Karaf Cellar from the sources, the requirements are:

Hardware:

- 100MB of free disk space for the Apache Karaf Cellar x.y source distributions or SVN checkout, the Maven build and the dependencies that Maven downloads.

Environment:

- Java SE Development Kit 1.7.x or greater ([http://www.oracle.com/technetwork/java/javase/]).
- Apache Maven 3.0.3 ([http://maven.apache.org/download.html]).

Note: Karaf Cellar requires Java 7 to compile, build and run.

2.3. Building on Windows

This procedure explains how to download and install the source distribution on a Windows system.

1. From a browser, navigate to http://karaf.apache.org/sub-projects/cellar/download.html
2. Select the desired distribution. For a source distribution, the filename will be similar to: {{apache-karaf-cellar-x.y-src.zip}}.
3. Extract Karaf Cellar from the ZIP file into a directory of your choice. Please remember the restrictions concerning illegal characters in Java paths, e.g. \!, % etc.
4. Build Karaf Cellar using Maven 3.0.3 or greater and Java 7.

The recommended method of building Karaf Cellar is the following:

```
cd [cellar_install_dir]\src
```

where `cellar_install_dir` is the directory in which Karaf Cellar was uncompressed.

```
mvn
```

Proceed to the Deploy Cellar section.

2.4. Building on Unix

This procedure explains how to download and install the source distribution on an Unix system.

1. From a browser, navigate to http://karaf.apache.org/sub-projects/cellar/download.html
2. Select the desired distribution. For a source distribution, the filename will be similar to: `apache-karaf-cellar-x.y-src.tar.gz`
3. Extract the files from the tarball file into a directory of your choice. For example:

```
gunzip apache-karaf-cellar-x.y-src.tar.gz
tar xvf apache-karaf-cellar-x.y-src.tar
```

Please remember the restrictions concerning illegal characters in Java paths, e.g. \!, % etc. . Build Karaf using Maven: The preferred method of building Karaf is the following:

```
cd [karaf_install_dir]/src
```

where `karaf_install_dir` is the directory in which Karaf Cellar was uncompressed.

```
mvn
```

Proceed to the Deploy Cellar section.

3. Deploy Cellar

This chapter describes how to deploy and start Cellar into a running Apache Karaf instance. This chapter assumes that you already know Apache Karaf basics, especially the notion of features and shell usage.

3.1. Registering Cellar features

Karaf Cellar is provided as a Karaf features XML descriptor.

Simply register the Cellar feature URL in your Karaf instance:

```
karaf@root()> feature:repo-add cellar
```

Now you have Cellar features available in your Karaf instance:

```
karaf@root()> feature:list |grep -i cellar
```

3.2. Starting Cellar

To start Cellar in your Karaf instance, you only need to install the Cellar feature:

```
karaf@root()> feature:install cellar
```

You can now see the Cellar components (bundles) installed:

```
karaf@root()> la|grep -i cellar
```

And Cellar cluster commands are now available:

```
karaf@root()> cluster:<TAB>
```

3.3. Optional features

Optionally, you can install additional features.

The cellar-event feature adds support of OSGi EventAdmin on the cluster:

```
karaf@root()> feature:install cellar-event
```

The cellar-obr feature adds support of OBR sync on the cluster:

```
karaf@root()> feature:install cellar-obr
```

The cellar-dosgi feature adds support of DOSGi (Distributed OSGi):

```
karaf@root()> feature:install cellar-dosgi
```

The cellar-cloud feature adds support of cloud blobstore, allowing to use instances located on a cloud provider:

```
karaf@root()> feature:install cellar-cloud
```

Please, see the sections dedicated to these features for details.

4. Core runtime and Hazelcast

Cellar uses Hazelcast as cluster engine.

When you install the cellar feature, a hazelcast feature is automatically installed, providing the `etc/hazelcast.xml` configuration file.

The `etc/hazelcast.xml` configuration file contains all the core configuration, especially: * the Hazelcast cluster identifiers (group name and password) * network discovery and security configuration

4.1. Hazelcast cluster identification

The `<group/>` element in the `etc/hazelcast.xml` defines the identification of the Hazelcast cluster:

```
    <group>
        <name>cellar</name>
        <password>pass</password>
    </group>
```

All Cellar nodes have to use the same name and password (to be part of the same Hazelcast cluster).

4.2. Network

The `<network/>` element in the etc/hazelcast.xml contains all the network configuration.

First, it defines the port numbers used by Hazelcast:

```
        <port auto-increment="true" port-count="100">5701</port>
        <outbound-ports>
            <!--
                Allowed port range when connecting to other nodes.
                0 or * means use system provided port.
            -->
            <ports>0</ports>
        </outbound-ports>
```

Second, it defines the mechanism used to discover the Cellar nodes: it's the `<join/>` element.

By default, Hazelcast uses unicast.

You can also use multicast (enabled by default in Cellar):

```
            <multicast enabled="true">
                <multicast-group>224.2.2.3</multicast-group>
                <multicast-port>54327</multicast-port>
            </multicast>
            <tcp-ip enabled="false"/>
            <aws enabled="false"/>
```

Instead of using multicast, you can also explicitly define the host names (or IP addresses) of the different Cellar nodes:

```
            <multicast enabled="false"/>
            <tcp-ip enabled="true"/>
            <aws enabled="false"/>
```

By default, it will bind to all interfaces on the node machine. It's possible to specify a interface:

```
            <multicast enabled="false"/>
            <tcp-ip enabled="true">
                <interface>127.0.0.1</interface>
            </tcp-ip>
            <aws enabled="false"/>
```

NB: in previous Hazelcast versions (especially the one used by Cellar 2.3.x), it was possible to have multicast and tcp-ip enabled in the same time. In Hazelcast 3.3.x (the version currently used by Cellar 3.0.x), only one discover mechanism can be enabled at a time. Cellar uses multicast by default (tcp-ip is disabled). If your network or network interface don't support multicast, you have to enable tcp-ip and disable multicast.

You can also discover nodes located on a Amazon instance:

```
            <multicast enabled="false"/>
            <tcp-ip enabled="false"/>
            <aws enabled="true">
                <access-key>my-access-key</access-key>
                <secret-key>my-secret-key</secret-key>
                <!--optional, default is us-east-1 -->
                <region>us-west-1</region>
                <!--optional, default is ec2.amazonaws.com. If set, region shouldn't be
set as it will override this property -->
                <host-header>ec2.amazonaws.com</host-header>
                <!-- optional, only instances belonging to this group will be discovered,
default will try all running instances -->
                <security-group-name>hazelcast-sg</security-group-name>
                <tag-key>type</tag-key>
                <tag-value>hz-nodes</tag-value>
            </aws>
```

Third, you can specific on which network interface the cluster is running (whatever the discovery mechanism used). By default, Hazelcast listens on all interfaces (0.0.0.0). But you can specify an interface:

```
        <interfaces enabled="true">
            <interface>10.10.1.*</interface>
        </interfaces>
```

Finally, you can also enable security transport on the cluster. Two modes are supported:

- SSL:

```
        <ssl enabled="true"/>
```

- Symmetric Encryption:

```
        <symmetric-encryption enabled="true">
            <!--
               encryption algorithm such as
               DES/ECB/PKCS5Padding,
               PBEWithMD5AndDES,
               AES/CBC/PKCS5Padding,
               Blowfish,
               DESede
            -->
            <algorithm>PBEWithMD5AndDES</algorithm>
            <!-- salt value to use when generating the secret key -->
            <salt>thesalt</salt>
            <!-- pass phrase to use when generating the secret key -->
            <password>thepass</password>
            <!-- iteration count to use when generating the secret key -->
            <iteration-count>19</iteration-count>
        </symmetric-encryption>
```

Cellar provides additional discovery mechanisms, See Discovery Service (jclouds and kubernetes) section for details.

5. Cellar nodes

This chapter describes the Cellar nodes manipulation commands.

5.1. Nodes identification

When you installed the Cellar feature, your Karaf instance became automatically a Cellar cluster node, and hence tries to discover the others Cellar nodes.

You can list the known Cellar nodes using the list-nodes command:

```
karaf@root()> cluster:node-list
  | Id              | Host Name | Port
-------------------------------------
x | node2:5702      | node2 | 5702
  | node1:5701      | node1 | 5701
```

The starting *x* indicates that it's the Karaf instance on which you are logged on (the local node).

NB: if you don't see the other nodes there (whereas they should be there), it's probably due to a network issue. By default, Cellar uses multicast to discover the nodes. If your network or network interface don't support multicast, you have to switch to tcp-ip instead of multicast. See Core Configuration section for details.

NB: in Cellar 2.3.x, Cellar used both multicast and tcp-ip by default. Due to a change in Hazelcast, it's no more possible to have both. Now, in Cellar 3.0.x, the default configuration is multicast enabled, tcp-ip disabled. See Core Configuration section for details.

5.2. Testing nodes

You can ping a node to test it:

```
karaf@root()> cluster:node-ping node1:5701
PING node1:5701
from 1: req=node1:5701 time=11 ms
from 2: req=node1:5701 time=12 ms
from 3: req=node1:5701 time=13 ms
from 4: req=node1:5701 time=7 ms
from 5: req=node1:5701 time=12 ms
```

5.3. Node Components: listener, producer, handler, consume, and synchronizer

A Cellar node is actually a set of components, each component is dedicated to a special purpose.

The `etc/org.apache.karaf.cellar.node.cfg` configuration file is dedicated to the configuration of the local node. It's where you can control the status of the different components.

5.4. Synchronizers and sync policy

A synchronizer is invoked when you:

- Cellar starts
- a node joins a cluster group (see link:groups for details about cluster groups)
- you explicitly call the `cluster:sync` command

We have a synchronizer per resource: feature, bundle, config, eventadmin (optional), obr (optional).

Cellar supports three sync policies:

- **cluster** (default): if the node is the first one in the cluster, it pushes its local state to the cluster, else if it's not the first node in the cluster, the node will update its local state with the cluster one (meaning that the cluster is the master).
- **node**: in this case, the node is the master, it means that the cluster state will be overwritten by the node state.
- **disabled**: in this case, it means that the synchronizer is not used at all, meaning the node or the cluster are not updated at all (at sync time).

You can configure the sync policy (for each resource, and each cluster group) in the `etc/org.apache.karaf.cellar.groups.cfg` configuration file:

```
default.bundle.sync = cluster
default.config.sync = cluster
default.feature.sync = cluster
default.obr.urls.sync = cluster
```

The `cluster:sync` command allows you to "force" the sync:

```
karaf@node1()> cluster:sync
Synchronizing cluster group default
        bundle: done
        config: done
        feature: done
        obr.urls: No synchronizer found for obr.urls
```

It's also possible to sync only a resource using:

- `-b` (`--bundle`) for bundle
- `-f` (`--feature`) for feature
- `-c` (`--config`) for configuration
- `-o` (`--obr`) for OBR URLs

or a given cluster group using the `-g` (`--group`) option.

5.5. Producer, consumer, and handlers

To notify the other nodes in the cluster, Cellar produces a cluster event.

For that, the local node uses a producer to create and send the cluster event. You can see the current status of the local producer using the `cluster:producer-status` command:

```
karaf@node1()> cluster:producer-status
  | Node             | Status
-----------------------------
x | 172.17.42.1:5701 | ON
```

The `cluster:producer-stop` and `cluster:producer-start` commands allow you to stop or start the local cluster event producer:

```
karaf@node1()> cluster:producer-stop
  | Node             | Status
-----------------------------
x | 172.17.42.1:5701 | OFF
karaf@node1()> cluster:producer-start
  | Node             | Status
-----------------------------
x | 172.17.42.1:5701 | ON
```

When the producer is off, it means that the node is "isolated" from the cluster as it doesn't send "outbound" cluster events to the other nodes.

On the other hand, a node receives the cluster events on a consumer. Like for the producer, you can see and control the consumer using dedicated command:

```
karaf@node1()> cluster:consumer-status
  | Node           | Status
---------------------------
x | localhost:5701 | ON
karaf@node1()> cluster:consumer-stop
  | Node           | Status
---------------------------
x | localhost:5701 | OFF
karaf@node1()> cluster:consumer-start
  | Node           | Status
---------------------------
x | localhost:5701 | ON
```

When the consumer is off, it means that node is "isolated" from the cluster as it doesn't receive "inbound" cluster events from the other nodes.

Different cluster events are involved. For instance, we have cluster event for feature, for bundle, for configuration, for OBR, etc. When a consumer receives a cluster event, it delegates the handling of the cluster event to a specific handler, depending of the type of the cluster event. You can see the different handlers and their status using the cluster:handler-status command:

```
karaf@node1()> cluster:handler-status
  | Node           | Status | Event Handler
-------------------------------------------------------------------------------
x | localhost:5701 | ON     | org.apache.karaf.cellar.config.ConfigurationEventHandler
x | localhost:5701 | ON     | org.apache.karaf.cellar.bundle.BundleEventHandler
x | localhost:5701 | ON     | org.apache.karaf.cellar.features.FeaturesEventHandler
```

You can stop or start a specific handler using the `cluster:handler-stop` and `cluster:handler-start` commands.

When a handler is stopped, it means that the node will receive the cluster event, but will not update the local resources dealt by the handler.

5.6. Listeners

The listeners are listening for local resource change.

For instance, when you install a feature (with `feature:install`), the feature listener traps the change and broadcast this change as a cluster event to other nodes.

To avoid some unexpected behaviors (especially when you stop a node), most of the listeners are switch off by default.

The listeners status are configured in the `etc/org.apache.karaf.cellar.node.cfg` configuration file.

NB: enabling listeners is at your own risk. We encourage you to use cluster dedicated commands and MBeans to manipulate the resources on the cluster.

6. Clustered resources

Cellar provides dedicated commands and MBeans for clustered resources.

Please, go into the cluster groups section for details.

7. Cellar groups

You can define groups in Cellar. A group allows you to define specific nodes and resources that are to be working together. This permits some nodes (those outside the group) not to need to sync'ed with changes of a node within a group.

By default, the Cellar nodes go into the default group:

```
karaf@root()> cluster:group-list
  | Group   | Members
------------------------------------------------
x | default | node2:5702 node1:5701(x)
```

The *x* indicates a local group. A local group is a group containing the local node (where we are connected).

7.1. New group

You can create a new group using the group-create command:

```
karaf@root()> cluster:group-create test
```

For now, the test group hasn't any nodes:

```
karaf@node1()> cluster:group-list
  | Group   | Members
------------------------------------------------
x | default | node2:5702 node1:5701(x)
  | test    |
```

7.2. Clustered Resources and Cluster Groups

7.2.1. Features

Cellar can manipulate features and features repositories on cluster groups.

You can use `cluster:feature-*` commands or the corresponding MBean for that.

You can list the features repositories on a given cluster group:

```
karaf@node1()> cluster:feature-repo-list default
Repository                  |    Located    | URL
-------------------------------------------------------------------------------------------------------------
jclouds-1.8.1               | cluster/local | mvn:org.apache.jclouds.karaf/jclouds-karaf/
1.8.1/xml/features
karaf-cellar-3.0.1-SNAPSHOT | cluster/local | mvn:org.apache.karaf.cellar/
apache-karaf-cellar/3.0.1-SNAPSHOT/xml/features
org.ops4j.pax.cdi-0.8.0     | cluster/local | mvn:org.ops4j.pax.cdi/pax-cdi-features/
0.8.0/xml/features
spring-3.0.2                | cluster/local | mvn:org.apache.karaf.features/spring/3.0.2/
xml/features
standard-3.0.2              | cluster/local | mvn:org.apache.karaf.features/standard/
3.0.2/xml/features
enterprise-3.0.2            | cluster/local | mvn:org.apache.karaf.features/enterprise/
3.0.2/xml/features
org.ops4j.pax.web-3.1.2     | cluster/local | mvn:org.ops4j.pax.web/pax-web-features/
3.1.2/xml/features
```

You have the name of the repository, and the URL, like in the feature:repo-list command. However, the cluster:feature-repo-list command provides the location of the features repository: * cluster means that the repository is defined only on the cluster group * local means that the repository is defined only on the local node (not on the cluster) * cluster/local means that the repository is defined both on the local node, but also on the cluster group

You can add a repository on a cluster group using the `cluster:feature-repo-add` command:

```
karaf@node1()> cluster:feature-repo-add default mvn:org.apache.activemq/activemq-karaf/
5.10.0/xml/features
```

You can remove a repository from a cluster group using the `cluster:feature-repo-remove` command:

```
karaf@node1()> cluster:feature-repo-remove default mvn:org.apache.activemq/activemq-karaf/
5.10.0/xml/features
```

You can list the features on a given cluster group:

```
karaf@node1()> cluster:feature-list default |more
Name                                | Version          | Installed | Located       |
Blocked
-------------------------------------------------------------------------------------------
gemini-blueprint                    | 1.0.0.RELEASE    |           | cluster/local |
package                             | 3.0.2            | x         | cluster/local |
jclouds-api-route53                 | 1.8.1            |           | cluster/local |
jclouds-rackspace-clouddns-uk       | 1.8.1            |           | cluster/local |
cellar-cloud                        | 3.0.1-SNAPSHOT   |           | local         |
in/out
webconsole                          | 3.0.2            |           | cluster/local |
cellar-shell                        | 3.0.1-SNAPSHOT   | x         | local         |
in/out
jclouds-glesys                      | 1.8.1            |           | cluster/local |
...
```

Like for the features repositories, you can note there the "Located" column containing where the feature is located (local to the node, or on the cluster group). You can also see the "Blocked" column indicating if the feature is blocked inbound or outbound (see the blocking policy).

You can install a feature on a cluster group using the `cluster:feature-install` command:

```
karaf@node1()> cluster:feature-install default eventadmin
```

You can uninstall a feature from a cluster group, using the `cluster:feature-uninstall` command:

```
karaf@node1()> cluster:feature-uninstall default eventadmin
```

Cellar also provides a feature listener, disabled by default as you can see in `etc/org.apache.karaf.cellar.node.cfg` configuration file:

```
feature.listener = false
```

The listener listens for the following local feature changes: * add features repository * remove features repository * install feature * uninstall feature

7.2.2. Bundles

Cellar can manipulate bundles on cluster groups.

You can use `cluster:bundle-*` commands or the corresponding MBean for that.

You can list the bundles in a cluster group using the `cluster:bundle-list` command:

```
karaf@node1()> cluster:bundle-list default |more
Bundles in cluster group default
ID | State    | Located       | Blocked | Version         | Name
-------------------------------------------------------------------------------------------
 0 | Active   | cluster/local |         | 2.2.0           | OPS4J Pax Url - aether:
 1 | Active   | cluster/local |         | 3.0.2           | Apache Karaf :: Deployer ::
Blueprint
 2 | Active   | cluster/local |         | 2.2.0           | OPS4J Pax Url - wrap:
 3 | Active   | cluster/local |         | 1.8.0           | Apache Felix Configuration
Admin Service
 4 | Active   | cluster/local |         | 3.0.2           | Apache Karaf :: Region :: Core
 ...
```

Like for the features, you can see the "Located" and "Blocked" columns.

You can install a bundle on a cluster group using the `cluster:bundle-install` command:

```
karaf@node1()> cluster:bundle-install default mvn:org.apache.servicemix.bundles/
org.apache.servicemix.bundles.commons-lang/2.4_6
```

You can start a bundle in a cluster group using the `cluster:bundle-start` command:

```
karaf@node1()> cluster:bundle-start default commons-lang
```

You can stop a bundle in a cluster group using the `cluster:bundle-stop` command:

```
karaf@node1()> cluster:bundle-stop default commons-lang
```

You can uninstall a bundle from a cluster group using the `cluster:bundle-uninstall` command:

```
karaf@node1()> cluster:bundle-uninstall default commons-lang
```

Like for the feature, Cellar provides a bundle listener disabled by default in `etc/org.apache.karaf.cellar.nodes.cfg`:

```
bundle.listener = false
```

The bundle listener listens the following local bundle changes: * install bundle * start bundle * stop bundle * uninstall bundle

7.2.3. Configurations

Cellar can manipulate configurations on cluster groups.

You can use `cluster:config-*` commands or the corresponding MBean for that.

You can list the configurations on a cluster group using the `cluster:config-list` command:

```
karaf@node1()> cluster:config-list default |more
----------------------------------------------------------------
Pid:            org.apache.karaf.command.acl.jaas
Located:        cluster/local
Blocked:
Properties:
   update = admin
   service.pid = org.apache.karaf.command.acl.jaas
----------------------------------------------------------------
...
```

You can note the "Blocked" and "Located" attributes, like for features and bundles.

You can list properties in a config using the `cluster:config-property-list` command:

```
karaf@node1()> cluster:config-property-list default org.apache.karaf.jaas
Property list for configuration PID org.apache.karaf.jaas for cluster group default
   encryption.prefix = {CRYPT}
   encryption.name =
   encryption.enabled = false
   encryption.suffix = {CRYPT}
   encryption.encoding = hexadecimal
   service.pid = org.apache.karaf.jaas
   encryption.algorithm = MD5
```

You can set or append a value to a config property using the `cluster:config-property-set` or `cluster:config-property-append` command:

```
karaf@node1()> cluster:config-property-set default my.config my.property my.value
```

You can delete a property in a config using the `cluster:config-property-delete` command:

```
karaf@node1()> cluster:config-property-delete default my.config my.property
```

You can delete the whole config using the `cluster:config-delete` command:

```
karaf@node1()> cluster:config-delete default my.config
```

Like for feature and bundle, Cellar provides a config listener disabled by default in `etc/org.apache.karaf.cellar.nodes.cfg`:

```
config.listener = false
```

The config listener listens the following local config changes: * create a config * add/delete/change a property * delete a config

As some properties may be local to a node, Cellar excludes some property by default. You can see the current excluded properties using the `cluster:config-property-excluded` command:

```
karaf@node1()> cluster:config-property-excluded
service.factoryPid, felix.fileinstall.filename, felix.fileinstall.dir,
felix.fileinstall.tmpdir, org.ops4j.pax.url.mvn.defaultRepositories
```

You can modify this list using the same command, or by editing the `etc/org.apache.karaf.cellar.node.cfg` configuration file:

```
#
# Excluded config properties from the sync
# Some config properties can be considered as local to a node, and should not be sync on
the cluster.
#
config.excluded.properties = service.factoryPid, felix.fileinstall.filename,
felix.fileinstall.dir, felix.fileinstall.tmpdir, org.ops4j.pax.url.mvn.defaultRepositories
```

7.2.4. OBR (optional)

See the OBR section for details.

7.2.5. EventAdmin (optional)

See the EventAdmin section for details.

7.3. Blocking policy

You can define a policy to filter the cluster events exchanges by the nodes (inbound or outbound).

It allows you to block or allow some resources on the cluster.

By adding a resource id in a blacklist, you block the resource. By adding a resource id in a whitelist, you allow the resource.

For instance, for feature, you can use the `cluster:feature-block` command to display or modify the current blocking policy for features:

```
karaf@node1()> cluster:feature-block default
INBOUND:
        whitelist: [*]
        blacklist: [config, cellar*, hazelcast, management]
OUTBOUND:
        whitelist: [*]
        blacklist: [config, cellar*, hazelcast, management]
```

NB: * is a wildcard.

You have the equivalent command for bundle and config:

```
karaf@node1()> cluster:bundle-block default
INBOUND:
        whitelist: [*]
        blacklist: [*.xml]
OUTBOUND:
        whitelist: [*]
        blacklist: [*.xml]
karaf@node1()> cluster:config-block default
INBOUND:
        whitelist: [*]
        blacklist: [org.apache.karaf.cellar*, org.apache.karaf.shell,
org.ops4j.pax.logging, org.ops4j.pax.web, org.apache.felix.fileinstall*,
org.apache.karaf.management, org.apache.aries.transaction]
OUTBOUND:
        whitelist: [*]
        blacklist: [org.apache.karaf.cellar*, org.apache.karaf.shell,
org.ops4j.pax.logging, org.ops4j.pax.web, org.apache.felix.fileinstall*,
org.apache.karaf.management, org.apache.aries.transaction]
```

Using those commands, you can also update the blacklist and whitelist for inbound or outbound cluster events.

8. OBR Support

Apache Karaf Cellar is able to "broadcast" OBR actions between cluster nodes of the same group.

8.1. Enable OBR support

To enable Cellar OBR support, the `cellar-obr` feature must first be installed:

```
karaf@root()> feature:install cellar-obr
```

The Cellar OBR feature will install the Karaf OBR feature which provides the OBR service (RepositoryAdmin).

8.2. Register repository URL in a cluster

The `cluster:obr-add-url` command registers an OBR repository URL (repository.xml) in a cluster group:

```
karaf@root()> cluster:obr-add-url group file:///path/to/repository.xml
karaf@root()> cluster:obr-add-url group http://karaf.cave.host:9090/cave/
repo-repository.xml
```

The OBR repository URLs are stored in a cluster distributed set. It allows new nodes to register the distributed URLs:

```
karaf@root()> cluster:obr-list-url group
file://path/to/repository.xml
http://karaf.cave.host:9090/cave/repo-repository.xml
```

When a repository is registered in the distributed OBR, Cave maintains a distributed set of bundles available on the OBR of a cluster group:

```
karaf@root()> cluster:obr-list group
Name                                                                                 | Symbolic
Name                                                                  | Version
-------------------------------------------------------------------------------------------------
Apache Aries JMX Blueprint Core                                                      |
org.apache.aries.jmx.blueprint.core                                   | 1.1.1.SNAPSHOT
Apache Karaf :: JAAS :: Command                                                      |
org.apache.karaf.jaas.command                                         | 2.3.6.SNAPSHOT
Apache Aries Proxy Service                                                           |
org.apache.aries.proxy.impl                                           | 1.0.3.SNAPSHOT
Apache Karaf :: System :: Shell Commands                                             |
org.apache.karaf.system.command                                       | 3.0.2.SNAPSHOT
Apache Karaf :: JDBC :: Core                                                         |
org.apache.karaf.jdbc.core                                            | 3.0.2.SNAPSHOT
Apache Aries Example SPI Provider Bundle 1                                           |
org.apache.aries.spifly.examples.provider1.bundle                     | 1.0.1.SNAPSHOT
Apache Aries Transaction Manager                                                     |
org.apache.aries.transaction.manager                                  | 1.1.1.SNAPSHOT
Apache Karaf :: Features :: Management                                               |
org.apache.karaf.features.management                                  | 2.3.6.SNAPSHOT
Apache Aries Blueprint Sample Fragment for Testing Annotation                        |
org.apache.aries.blueprint.sample-fragment                            | 1.0.1.SNAPSHOT
Apache Karaf :: Management :: MBeans :: OBR                                          |
org.apache.karaf.management.mbeans.obr                                | 2.3.6.SNAPSHOT
Apache Karaf :: JNDI :: Core                                                         |
org.apache.karaf.jndi.core                                            | 2.3.6.SNAPSHOT
Apache Karaf :: Shell :: SSH                                                         |
org.apache.karaf.shell.ssh                                            | 3.0.2.SNAPSHOT
Apache Aries Blueprint Web OSGI                                                      |
org.apache.aries.blueprint.webosgi                                    | 1.0.2.SNAPSHOT
Apache Aries Blueprint JEXL evaluator                                                |
org.apache.aries.blueprint.jexl.evaluator                             | 1.0.1.SNAPSHOT
Apache Karaf :: JDBC :: Command                                                      |
org.apache.karaf.jdbc.command                                         | 3.0.2.SNAPSHOT
...
```

When you remove a repository URL from the distributed OBR, the bundles' distributed set is updated:

```
karaf@root()> cluster:obr-remove-url group http://karaf.cave.host:9090/cave/
repo-repository.xml
```

8.3. Deploying bundles using the cluster OBR

You can deploy a bundle (and that bundle’s dependent bundles) using the OBR on a given cluster group:

```
karaf@root()> cluster:obr-deploy group bundleId
```

The bundle ID is the symbolic name, viewable using the `cluster:obr-list` command. If you don't provide the version, the OBR deploys the latest version available. To provide the version, use a comma after the symbolic name:

```
karaf@root()> cluster:obr-deploy group
org.apache.servicemix.specs.java-persistence-api-1.1.1
karaf@root()> cluster:obr-deploy group org.apache.camel.camel-jms,2.9.0.SNAPSHOT
```

The OBR will automatically install the bundles required to satisfy the bundle dependencies.

The deploy command doesn't start bundles by default. To start the bundles just after deployment, you can use the `-s` option:

```
karaf@root()> cluster:obr-deploy -s group org.ops4j.pax.web.pax-web-runtime
```

9. OSGi Event Broadcasting support (eventadmin)

Apache Karaf Cellar is able to listen all OSGi events on the cluster nodes, and broadcast each events to other nodes.

9.1. Enable OSGi Event Broadcasting support

OSGi Event Broadcasting is an optional feature. To enable it, you have to install the cellar-eventadmin feature:

```
karaf@root()> feature:install cellar-eventadmin
```

9.2. OSGi Event Broadcast in action

As soon as the cellar-eventadmin feature is installed (on all nodes that should use the clustered eventadmin), Cellar listens all OSGi events, and broadcast these events to all nodes of the same cluster group.

10. HTTP Balancer

Apache Karaf Cellar is able to expose servlets local to a node on the cluster. It means that a client (browser) can use any node in the cluster, proxying the requests to the node actually hosting the servlets.

10.1. Enable HTTP Balancer

To enable Cellar HTTP Balancer, you have to first install the `http` and `http-whiteboard` features:

```
karaf@root()> feature:install http
karaf@root()> feature:install http-whiteboard
```

Now, we install the `cellar-http-balancer` feature, actually providing the balancer:

```
karaf@root()> feature:install cellar-http-balancer
```

Of course, you can use Cellar to spread the installation of the `cellar-http-balancer` feature on all nodes in the cluster group:

```
karaf@root()> cluster:feature-install default cellar-http-balancer
```

It's done: the Cellar HTTP Balancer is now enabled. It will expose proxy servlets on nodes.

10.2. Balancer in action

To illustrate Cellar HTTP Balancer in action, you need at least a cluster with two nodes.

On node1, we enable the Cellar HTTP Balancer:

```
karaf@node1()> feature:install http
karaf@node1()> feature:install http-whiteboard
karaf@node1()> feature:repo-add cellar 4.0.0
karaf@node1()> feature:install cellar
karaf@node1()> cluster:feature-install default cellar-http-balancer
```

Now, we install the webconsole on node1:

```
karaf@node1()> feature:install webconsole
```

We can see the "local" servlets provided by the webconsole feature using the `http:list` command:

```
karaf@node1()> http:list
ID  | Servlet          | Servlet-Name    | State       | Alias               | Url
-------------------------------------------------------------------------------------------------
101 | KarafOsgiManager | ServletModel-2  | Undeployed  | /system/console     | [/system/
console/*]
103 | GogoPlugin       | ServletModel-7  | Deployed    | /gogo               | [/gogo/*]
102 | FeaturesPlugin   | ServletModel-6  | Deployed    | /features           |
[/features/*]
101 | ResourceServlet  | /res            | Deployed    | /system/console/res | [/system/
console/res/*]
101 | KarafOsgiManager | ServletModel-11 | Deployed    | /system/console     | [/system/
console/*]
105 | InstancePlugin   | ServletModel-9  | Deployed    | /instance           |
[/instance/*]
```

You can access to the webconsole using a browser on http://localhost:8181/system/console.

We can see that Cellar HTTP Balancer exposed the servlets to the cluster, using the `cluster:http-list` command:

```
karaf@node1()> cluster:http-list default
Alias               | Locations
--------------------------------------------------------------------
/system/console/res | http://172.17.42.1:8181/system/console/res
/gogo               | http://172.17.42.1:8181/gogo
/instance           | http://172.17.42.1:8181/instance
/system/console     | http://172.17.42.1:8181/system/console
/features           | http://172.17.42.1:8181/features
```

On another node (node2), we install `http`, `http-whiteboard` and `cellar` features:

```
karaf@node1()> feature:install http
karaf@node1()> feature:install http-whiteboard
karaf@node1()> feature:repo-add cellar 4.0.0
karaf@node1()> feature:install cellar
```

WARNING

> if you run the nodes on a single machine, you have to provision `etc/org.ops4j.pax.web.cfg` configuration file containing the `org.osgi.service.http.port property` with a port number different to 8181. For this example, we use the following `etc/org.ops4j.pax.web.cfg` file:

```
org.osgi.service.http.port=8041
```

On node1, as we installed the `cellar-http-balancer` using `cluster:feature-install` command, it's automatically installed when node2 joins the default cluster group.

We can see the HTTP endpoints available on the cluster using the `cluster:http-list` command:

```
karaf@node2()> cluster:http-list default
Alias               | Locations
--------------------------------------------------------------
/system/console/res | http://172.17.42.1:8181/system/console/res
/gogo               | http://172.17.42.1:8181/gogo
/instance           | http://172.17.42.1:8181/instance
/system/console     | http://172.17.42.1:8181/system/console
/features           | http://172.17.42.1:8181/features
```

If we take a look on the HTTP endpoints locally available on node2 (using `http:list` command), we can see the proxies created by Cellar HTTP Balancer:

```
karaf@node2()> http:list
ID  | Servlet                   | Servlet-Name   | State       | Alias               |
Url
-------------------------------------------------------------------------------------------------------
100 | CellarBalancerProxyServlet | ServletModel-3 | Deployed    | /gogo               |
[/gogo/*]
100 | CellarBalancerProxyServlet | ServletModel-2 | Deployed    | /system/console/res |
[/system/console/res/*]
100 | CellarBalancerProxyServlet | ServletModel-6 | Deployed    | /features           |
[/features/*]
100 | CellarBalancerProxyServlet | ServletModel-5 | Deployed    | /system/console     |
[/system/console/*]
100 | CellarBalancerProxyServlet | ServletModel-4 | Deployed    | /instance           |
[/instance/*]
```

You can use a browser on http://localhost:8041/system/console: you will actually use the webconsole from node1, as Cellar HTTP Balancer proxies from node2 to node1.

Cellar HTTP Balancer randomly chooses one endpoint providing the HTTP endpoint.

11. HTTP Session Replication

Apache Karaf Cellar supports replication of the HTTP sessions on the cluster.

It means that the same web application deployed on multiple nodes in the cluster will share the same HTTP sessions pool, allowing clients to transparently connect to any node, without loosing any session state.

11.1. Enable Cluster HTTP Session Replication

In order to be able to be stored on the cluster, all HTTP Sessions used in your web application have to implement Serializable interface. Any non-serializable attribute has to be flagged as transient.

You have to enable a specific filter in your application to enable the replication. See next chapter for details.

At runtime level, you just have to install `http`, `http-whiteboard`, and `cellar` features:

```
karaf@root()> feature:install http
karaf@root()> feature:install http-whiteboard
karaf@root()> feature:repo-add cellar
karaf@root()> feature:install cellar
```

11.2. Web Application Session Replication

In order to use HTTP session replication on the cluster, you just have to add a filter in your web application.

Basically, the `WEB-INF/web.xml` file of your web application should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_3_0.xsd"
        version="3.0">

        <filter>
                <filter-name>hazelcast-filter</filter-name>
                <filter-class>com.hazelcast.web.WebFilter</filter-class>
            <!--
                Name of the distributed map storing
                your web session objects
            -->
                <init-param>
                        <param-name>map-name</param-name>
                        <param-value>my-sessions</param-value>
                </init-param>
            <!-- How is your load-balancer configured? stick-session means all requests of
                    a session is routed to the node where the session is first created.
This is
                    excellent for performance. If sticky-session is set to false, when a
session
                     is updated on a node, entry for this session on all other nodes is
invalidated.
                     You have to know how your load-balancer is configured before setting
this
                     parameter. Default is true. -->
                <init-param>
                        <param-name>sticky-session</param-name>
                        <param-value>false</param-value>
                </init-param>
            <!--
                Are you debugging? Default is false.
            -->
                <init-param>
                        <param-name>debug</param-name>
                        <param-value>false</param-value>
                </init-param>
        </filter>
        <filter-mapping>
                <filter-name>hazelcast-filter</filter-name>
                <url-pattern>/*</url-pattern>
                <dispatcher>FORWARD</dispatcher>
                <dispatcher>INCLUDE</dispatcher>
                <dispatcher>REQUEST</dispatcher>
        </filter-mapping>
        <listener>
                <listener-class>com.hazelcast.web.SessionListener</listener-class>
```

```
        </listener>

</web-app>
```

12. DOSGi and Transport

DOSGi (Distributed OSGi) enables the distribution of OSGi services across the Cellar nodes.

The purpose of the Cellar DOSGi is to leverage the Cellar resources (Hazelcast instances, distributed map, etc), and to be very easy to use.

DOSGi is provided by installing the optional feature `cellar-dosgi`.

To be available and visible for the others nodes, the OSGi service should only have the service.exported.interfaces property:

```
<service ref="MyService" interface="my.interface">
  <service-properties>
    <entry key="service.exported.interfaces" value="*"/>
  </service-properties>
</service>
```

You can see all OSGi services "flagged" as distributed (available for the nodes) using the `cluster:list-service` command:

```
karaf@root()> cluster:service-list
```

A "client" bundle could use this service. If the service is not available locally, Cellar will "route" the service call to the remote remote containing the service.

13. Discovery Services

The Discovery Services allow you to use third party libraries to discover the nodes member of the Cellar cluster.

13.1. jClouds

Cellar relies on Hazelcast (http://www.hazelcast.com) in order to discover cluster nodes. This can happen either by using unicast, multicast or specifying the ip address of each node. See the Core Configuration section for details.

Unfortunately multicast is not allowed in most IaaS providers and the alternative of specifying all IP addresses creates maintenance difficulties, especially since in most cases the addresses are not known in advance.

Cellar solves this problem using a cloud discovery service powered by jclouds (http://jclouds.apache.org).

13.1.1. Cloud discovery service

Most cloud providers provide cloud storage among other services. Cellar uses the cloud storage via jclouds, in order to determine the IP addresses of each node so that Hazelcast can find them.

This approach is also called blackboard and refers to the process where each node registers itself in a common storage are so that other nodes know its existence.

13.1.2. Installing Cellar cloud discovery service

To install the cloud discovery service simply install the appropriate jclouds provider and then install cellar-cloud feature. Amazon S3 is being used here for this example, but the below applies to any provider supported by jclouds.

```
karaf@root()> feature:install jclouds-aws-s3
karaf@root()> feature:install cellar-cloud
```

Once the feature is installed, you're required to create a configuration that contains credentials and the type of the cloud storage (aka blobstore). To do that add a configuration file under the etc folder with the name `org.apache.karaf.cellar.cloud-<provider>.cfg` and place the following information there:

```
provider=aws-s3 (this varies according to the blobstore provider)
identity=<the identity of the blobstore account>
credential=<the credential/password of the blobstore account)
container=<the name of the bucket>
validity=<the amount of time an entry is considered valid, after that time the entry is
removed>
```

For instance, you can create `etc/org.apache.karaf.cellar.cloud-mycloud.cfg` containing:

```
provider=aws-s3
identity=username
credential=password
container=cellar
validity=360000
```

NB: you can find the cloud providers supported by jclouds here http://repo1.maven.org/maven2/org/apache/jclouds/provider/. You have to install the corresponding jclouds feature for the provider.

After creating the file the service will check for new nodes. If new nodes are found the Hazelcast instance configuration will be updated and the instance restarted.

13.2. Kubernetes & docker.io

Kubernetes (http://kubernetes.io) is an open source orchestration system for docker.io containers. It handles scheduling onto nodes in a compute cluster and actively manages workloads to ensure that their state matches the users declared intentions.

Using the concepts of "labels", "pods", "replicationControllers" and "services", it groups the containers which make up an application into logical units for easy management and discovery.

Following the aforementioned concept will most likely change how you package and provision your Karaf based applications. For instance, you will eventually have to provide a Docker image with a pre-configured Karaf, KAR files in deployment folder, etc. so that your Kubernetes container may bootstrap everything on boot.

The Cellar Kubernetes discovery service is a great complement to the Karaf docker.io feature (allowing you to easily create and manage docker.io images in and for Karaf).

13.2.1. Kubernetes discovery service

In order to determine the IP address of each node, so that Hazelcast can connect to them, the Kubernetes discovery service queries the Kubernetes API for containers labeled with the `pod.label.key` and `pod.label.key` specified in `etc/org.apache.karaf.cellar.kubernetes-name.cfg`.

The name in `etc/org.apache.karaf.cellar.kubernetes-name.cfg` is a name of the choice. It allows you to create multiple Kubernetes discovery services. Thanks to that, the Cellar nodes can be discovered on different Kubernetes.

So, you **must be sure** to label your containers (pods) accordingly.

After a Cellar node starts up, Kubernetes discovery service will configure Hazelcast with currently running Cellar nodes. Since Hazelcast follows a peer-to-peer all-shared topology, whenever nodes come up and down, the cluster will remain up-to-date.

13.2.2. Installing Kubernetes discovery service

To install the Kubernetes discovery service, simply install cellar-kubernetes feature.

```
karaf@root()> feature:install cellar-kubernetes
```

Once the cellar-kubernetes feature is installed, you have to create the Kubernetes provider configuration file. If you have multiple Kubernetes instances, you create one configuration file per instance.

For instance, you can create `etc/org.apache.karaf.cellar.kubernetes-myfirstcluster.cfg` containing:

```
host=localhost
port=8080
pod.label.key=name
pod.label.value=cellar
```

and another one `etc/org.apache.karaf.cellar.kubernetes-mysecondcluster.cfg` containing:

```
host=192.168.134.2
port=8080
pod.label.key=name
pod.label.value=cellar
```

In case you change the file, the discovery service will check again for new nodes. If new nodes are found, Hazelcast configuration will be updated and the instance restarted.

Architecture Guide

1. Architecture Overview

The core concept behind Karaf Cellar is that each node can be a part of one or more groups that provide the node distributed memory for keeping data (e.g. configuration, features information, other) and a topic which is used to exchange events with the rest of the group nodes.

architecture

Each group comes with a configuration, which defines which events are to be broadcasted and which are not. Whenever a local change occurs to a node, the node will read the setup information of all the groups that it belongs to and broadcasts the event to the groups that are whitelisted to the specific event.

The broadcast operation happens via a distributed topic provided by the group. For the groups that the broadcast reaches, the distributed configuration data will be updated so that nodes that join in the future can pickup the change.

2. Supported Events

There are 3 types of events:

- Configuration change event.
- Features repository added/removed event.
- Features installed/uninstalled event.

Optionally (by installing the corresponding features), Cellar supports the following additional events:

- EventAdmin
- OBR

For each of the event types above a group may be configured to enabled synchronization, and to provide a whitelist/blacklist of specific event IDs.

For instance, the default group is configured to allow synchronization of configuration. This means that whenever a change occurs via the config admin to a specific PID, the change will pass to the distributed memory of the default group and will also be broadcasted to all other default group nodes using the topic.

This happens for all PIDs but not for org.apache.karaf.cellar.node which is marked as blacklisted and will never be written or read from the distributed memory, nor will be broadcasted via the topic.

The user can add/remove any PID he wishes to the whitelist/blacklist.

3. The role of Hazelcast

The idea behind the clustering engine is that for each unit that we want to replicate, we create an event, broadcast the event to the cluster and hold the unit state to a shared resource, so that the rest of the nodes can look up and retrieve the changes.

shared architecture

For instance, we want all nodes in our cluster to share configuration for PIDs a.b.c and x.y.z. On node "Karaf A" a change occurs on a.b.c. "Karaf A" updated the shared repository data for a.b.c and then notifies the rest of the nodes that a.b.c has changed. Each node looks up the shared repository and retrieves changes.

The architecture as described so far could be implemented using a database/shared filesystem as a shared resource and polling instead of multicasting events. So why use Hazelcast ?

Hazelcast fits in perfectly because it offers:

- Auto discovery
  - Cluster nodes can discover each other automatically.
  - No configuration is required.
- No single point of failure
  - No server or master is required for clustering
  - The shared resource is distributed, hence we introduce no single point of failure.
- Provides distributed topics

- Using in memory distributed topics allows us to broadcast events/commands which are valuable for management and monitoring.

In other words, Hazelcast allows us to setup a cluster with zero configuration and no dependency to external systems such as a database or a shared file system.

See the Hazelcast documentation at http://www.hazelcast.com/documentation.jsp for more information.

4. Design

The design works with the following entities:

- **OSGi Listener** is an interface which implements a listener for specific OSGi events (e.g. `ConfigurationListener`).
- **Event** is the object that contains all the required information required to describe the event (e.g. PID changed).
- **Event Topic** is the distributed topic used to broadcast events. It is common for all event types.
- **Shared Map** is the distributed collection that serves as shared resource. We use one per event type.
- **Event Handler** is the processor which processes remote events received through the topic.
- **Event Dispatcher** is the unit which decides which event should be processed by which event handlers.
- **Command** is a special type of event that is linked to a list of events that represent the outcome of the command.
- **Result** is a special type of event that represents the outcome of a command. Commands and results are correlated.

event flow

The OSGi specification uses the `Events` and `Listener` paradigms in many situations (e.g. `ConfigurationChangeEvent` and `ConfigurationListener`). By implementing such a Listener and exposing it as an OSGi service to the Service Registry, we can be sure that we are "listening" for the events that we are interested in.

When the listener is notified of an event, it forwards the `Event` object to a Hazelcazst distributed topic. To keep things as simple as possible, we keep a single topic for all event types. Each node has a listener registered on that topic and gets/sends all events to the event dispatcher.

When the Event Dispatcher receives an event, it looks up an internal registry (in our case the OSGi Service Registry) to find an Event Handler that can handle the received Event. The handler found receives the event and processes it.

5. Broadcasting commands

Commands are a special kind of event. They imply that when they are handled, a `Result` event will be fired containing the outcome of the command. For each command, we have one result per recipient.

Each command contains an unique id (unique for all cluster nodes, created from Hazelcast). This id is used to correlate the request with the result. For each result successfully correlated the result is added to list of results on the command object. If the list gets full of if 10 seconds from the command execution have elapsed, the list is moved to a blocking queue from which the result can be retrieved.

The following code snippet shows what happens when a command is sent for execution:

```
public Map<node,result> execute(Command command) throws Exception {
   if (command == null) {
      throw new Exception("Command store not found");
   } else {
      //store the command to correlate it with the result.
      commandStore.getPending().put(command.getId(), command);
      //I create a timeout task and schedule it
      TimeoutTask timeoutTask = new TimeoutTask(command, commandStore);
      ScheduledFuture timeoutFuture = timeoutScheduler.schedule(timeoutTask,
command.getTimeout(), TimeUnit.MILLISECONDS);
   }
   if (producer != null) {
      //send the command to the topic
      producer.produce(command);
      //retrieve the result list from the blocking queue.
      return command.getResult();
   }
   throw new Exception("Command producer not found");
}
```

Last updated 2015-12-10 18:03:53 CET