



Apache Karaf Cellar  
Version 2.2.2  
\\\\\\\\\\\\\\\\Apache Karaf CellarUser Guide\\\\\\\\\\\\\\\\

Copyright 2011 The Apache Software Foundation

# Table of contents

Overview  
User Guide  
Architecture Guide

# Overview

# Karaf Cellar Overview

Apache Karaf Cellar is a Apache Karaf sub-project which provides clustering support between Karaf instances.

Cellar allows you to manage a cluster of several Karaf instances, providing synchronization between instances.

Here is a short list of Cellar features:

- **Discovery:** when you install Cellar into a Karaf instance, it automatically tries to join the cluster of other Cellar -running Karaf instances it discovers. There is no configuration required to join the cluster, the discovery is made behind the scenes, which multicast or unicast used for discovery.
- **Cluster Group:** a Karaf node can be part of one or more cluster groups. In Cellar, you can define cluster groups per your requirements. Resources will be sync'ed between members of the same group.
- **Distributed Configuration Admin:** Cellar distributes configuration data, both of Cellar-specific and Karaf etc/\*.cfg configuration files. The distribution is event driven and filtered by group. You can tune the configuration replication using blacklists/whitelists on the configuration ID (PID).
- **Distributed Features Service:** Cellar distributes the features and features repository information, also an event-driven process.
- **Provisioning:** Cellar provides shell commands for basic provisioning. It can also use an OBR backend or another provisioning tool such as Apache ACE.



# Installation

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

## PRE-INSTALLATION REQUIREMENTS

As Cellar is a Karaf sub-project, you need a running Karaf instance.

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

## 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 1.6.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 6 to compile, build and run.

## 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 6.  
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
```

5. Proceed to the Deploy Cellar chapter.

## **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.

4. 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
```

5. Proceed to the Deploy Cellar chapter.

# 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.

## 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> features:addurl mvn:org.apache.karaf.cellar/  
apache-karaf-cellar/2.2.1/xml/features
```

Now you have Cellar features available in your Karaf instance:

```
karaf@root> features:list|grep -i cellar  
[uninstalled] [2.2.1          ]  
cellar  
Karaf clustering  
[uninstalled] [2.2.1          ]  
cellar-webconsole  
Karaf Cellar Webconsole Plugin
```

## STARTING CELLAR

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

```
karaf@root> features:install cellar
```

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

```
karaf@root> ls|grep -i cellar  
[ 56] [Active      ] [Created      ] [      ] [ 60] Apache  
Karaf :: Cellar :: Core (2.2.1)  
[ 57] [Active      ] [Created      ] [      ] [ 60] Apache  
Karaf :: Cellar :: Config (2.2.1)
```



```

[ 58] [Active      ] [Created      ] [      ] [ 60] Apache
Karaf :: Cellar :: Features (2.2.1)
[ 59] [Active      ] [Created      ] [      ] [ 60] Apache
Karaf :: Cellar :: Bundle (2.2.1)
[ 60] [Active      ] [Created      ] [      ] [ 60] Apache
Karaf :: Cellar :: Utils (2.2.1)
[ 61] [Active      ] [Created      ] [      ] [ 60] Apache
Karaf :: Cellar :: Shell (2.2.1)
[ 62] [Active      ] [      ] [      ] [ 60] Apache
Karaf :: Cellar :: Hazelcast (2.2.1)

```

And Cellar cluster commands are now available:

```

karaf@root> cluster:<TAB>
cluster:config-list           cluster:config-proplist
cluster:config-propset       cluster:consumer-start
cluster:consumer-status      cluster:consumer-stop
cluster:features-install     cluster:features-list
cluster:features-uninstall   cluster:group-create
cluster:group-delete         cluster:group-join
cluster:group-list           cluster:group-quit
cluster:group-set            cluster:handler-start
cluster:handler-status       cluster:handler-stop
cluster:list-nodes           cluster:ping
cluster:producer-start       cluster:producer-status
cluster:producer-stop

```

# Cellar nodes

This chapter describes the Cellar nodes manipulation commands.

## 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:list-nodes
  No. Host Name          Port ID
*   1 node1.local      5701 node1.local:5701
    2 node2.local      5702 node2.local:5702
```

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

## TESTING NODES

You can ping a node to test it:

```
karaf@root> cluster:ping node2.local:5702
Pinging node :node2.local:5702
PING 1 node2.local:5702 82ms
PING 2 node2.local:5702 11ms
PING 3 node2.local:5702 14ms
```

## NODES SYNC

Cellar allows nodes to 'sync' state. It currently covers features, configs, and bundles.

For instance, if you install a feature (eventadmin for example) on node1:

```
karaf@node1> features:install eventadmin
karaf@node1> features:list|grep -i eventadmin
[installed ] [2.2.1 ] eventadmin                                karaf-2.2.1
```

You can see that the eventadmin feature has been installed on node2:

```
karaf@node2> features:list|grep -i eventadmin  
[installed ] [2.2.1 ] eventadmin karaf-2.2.1
```

Features uninstall works in the same way. Basically, Cellar synchronisation is completely transparent.

Configuration is also synchronized.

# 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'd with changes of a node within a group.

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

```
karaf@root> cluster:group-list
Node                Group
* node1.local:5701  default
  node2.local:5702  default
```

As for node, the starting \* shows the local node/group.

## NEW GROUP

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

```
karaf@root> cluster:group-create test
Name                test
Members             []
```

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

```
kaaf@root> cluster:group-list
Node                Group
  node1.local:5701  default
* node2.local:5702  default
  test
```

## GROUP CONFIGURATION

You can see the configuration PID associated with a given group, for instance the default group:

```
karaf@root> cluster:config-list default
PIDs for group:default
PID
org.apache.felix.fileinstall.3e4e22ea-8495-4612-9839-a537c8a7a503
org.apache.felix.fileinstall.1afcd688-b051-4b12-a50e-97e40359b24e
org.apache.karaf.features
org.apache.karaf.log
org.apache.karaf.features.obr
org.ops4j.pax.logging
org.apache.karaf.cellar.groups
org.ops4j.pax.url.mvn
org.apache.karaf.jaas
org.apache.karaf.shell
```

You can use the `cluster:config-proplist` and `config-propset` commands to list, add and edit the configuration.

For instance, in the test group, we don't have any configuration:

```
karaf@root> cluster:config-list test
No PIDs found for group:test
```

We can create a `tstcfg` config in the test group, containing `name=value` property:

```
karaf@root> cluster:config-propset test tstcfg name value
```

Now, we have this property in the test group:

```
karaf@root> cluster:config-list test
PIDs for group:test
PID
tstcfg
karaf@root> cluster:config-proplist test tstcfg
Property list for PID:tstcfg for group:test
Key                                     Value
name                                   value
```

## GROUP MEMBERS

You can define a node member of one of more group:

```
karaf@root> cluster:group-join test node1.local:5701
Node                                     Group
```

```
node1:5701 default
* node2:5702 default
node1:5701 test
```

The node can be local or remote.

Now, the members of a given group will inherit of all configuration defined in the group. This means that node1 now knows the tstcfg configuration because it's a member of the test group:

```
karaf@root> config:edit tstcfg
karaf@root> proplist
  service.pid = tstcfg
  name = value
```

## GROUP FEATURES

Configuration and features can be assigned to a given group.

```
karaf@root> cluster:features-list default
Features for group:default
```

Name	Version
Status	
spring-dm	1.2.1
true	
kar	2.2.1
false	
config	2.2.1
true	
http-whiteboard	2.2.1
false	
application-without-isolation	0.3
false	
war	2.2.1
false	
standard	2.2.1
false	
management	2.2.1
true	
http	2.2.1
false	
transaction	0.3
false	

jetty	7.4.2.v20110526
false	
wrapper	2.2.1
false	
jndi	0.3
false	
obr	2.2.1
false	
jpa	0.3
false	
webconsole-base	2.2.1
false	
hazelcast	1.9.3
true	
eventadmin	2.2.1
false	
spring-dm-web	1.2.1
false	
ssh	2.2.1
true	
spring-web	3.0.5.RELEASE
false	
hazelcast-monitor	1.9.3
false	
jaspyt-encryption	2.2.1
false	
webconsole	2.2.1
false	
spring	3.0.5.RELEASE
true	

karaf@root> cluster:features-list test

Features for group:test

Name	Version
Status	
webconsole	2.2.1
false	
spring-dm	1.2.1
true	
eventadmin	2.2.1
false	
http	2.2.1
false	

war	2.2.1
false	
http-whiteboard	2.2.1
false	
obr	2.2.1
false	
spring	3.0.5.RELEASE
true	
hazelcast-monitor	1.9.3
false	
webconsole-base	2.2.1
false	
management	2.2.1
true	
hazelcast	1.9.3
true	
jpa	0.3
false	
jndi	0.3
false	
standard	2.2.1
false	
jetty	7.4.2.v20110526
false	
application-without-isolation	0.3
false	
config	2.2.1
true	
spring-web	3.0.5.RELEASE
false	
wrapper	2.2.1
false	
transaction	0.3
false	
spring-dm-web	1.2.1
false	
ssh	2.2.1
true	
jaspyt-encryption	2.2.1
false	
kar	2.2.1
false	



Now we can "install" a feature for a given cluster group:

```
karaf@root> cluster:features-install test eventadmin
karaf@root> cluster:features-list test|grep -i event
eventadmin                2.2.1 true
```

Below, we see that the eventadmin feature has been installed on this member of the test group:

```
karaf@root> features:list|grep -i event
[installed ] [2.2.1 ] eventadmin
karaf-2.2.1
```

# Architecture Guide

# 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 group members.

Each group comes with a configuration, which defines which events are to be broadcast 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 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.

# Supported Events

There are 3 types of events:

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

For each of the event types above a group may be configured to enable 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 members 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.

# 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.

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.

# Design

The design works with the following entities:

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

The OSGi specification uses the Events and Listener paradigm in many situations (e.g. ConfigurationChangeEvent and ConfigurationListener). By implementing such 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 Hazelcast 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.

# Broadcasting commands

Commands are a special kind of events. 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 or 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");
}
```