Apache Karaf
Version 4.0.3

Apache Karaf
Users' Guide

The PDF format of the Karaf Manual has been generated by Prince XML
(http://www.princexml.com).

Table of contents

Overview

Apache Karaf Overview

Apache Karaf is a modern and polymorphic container.

It's a lightweight, powerful, and enterprise ready container powered by OSGi. You can deploy different kind of applications in Karaf, OSGi or non-OSGi.

With this flexibility, Karaf is the perfect container for microservices, systems integration, big data, and much more.

Apache Karaf uses either the Apache Felix or Eclipse Equinox OSGi frameworks, providing additional features on top of the framework.

Apache Karaf can be scaled from a very lightweight container to a fully featured enterprise service: it's a very flexible and extensible container, covering all the major needs.

Here is a short list of provided features:

- **Hot deployment**: simply drop a file in the `deploy` directory, Apache Karaf will detect the type of the file and try to deploy it.
- **Complete Console**: Apache Karaf provides a complete Unix-like console where you can completely manage the container.
- **Dynamic Configuration**: Apache Karaf provides a set of commands focused on managing its own configuration. All configuration files are centralized in the `etc` folder. Any change in a configuration file is noticed and reloaded.
- **Advanced Logging System**: Apache Karaf supports all the popular logging frameworks (slf4j, log4j, etc). Whichever logging framework you use, Apache Karaf centralizes the configuration in one file.
- **Provisioning**: Apache Karaf supports a large set of URLs where you can install your applications (Maven repository, HTTP, file, etc). It also provides the concept of "Karaf Features" which is a way to describe your application.
- **Management**: Apache Karaf is an enterprise-ready container, providing many management indicators and operations via JMX.
- **Remote**: Apache Karaf embeds an SSHd server allowing you to use the console remotely. The management layer is also accessible remotely.
- **Security**: Apache Karaf provides a complete security framework (based on JAAS), and provides a RBAC (Role-Based Access Control) mechanism for console and JMX access.

- **Instances**: multiple instances of Apache Karaf can be managed directly from a main instance (root).
- **OSGi frameworks**: Apache Karaf is not tightly coupled to one OSGi framework. By default, Apache Karaf runs with the Apache Felix Framework, but you can easily switch to Equinox (just change one property in a configuration file).

Quick Start

Quick Start

These instructions should help you get Apache Karaf up and running in 5 to 15 minutes.

PREREQUISITES

Karaf requires a Java SE 7 or Java SE 8 environment to run. Refer to http://www.oracle.com/technetwork/java/javase/ for details on how to download and install Java SE 1.7 or greater.

- Open a Web browser and access the following URL: http://karaf.apache.org/index/community/download.html
- Download the binary distribution that matches your system (zip for windows, tar.gz for unixes)
- Extract the archive to a new folder on your hard drive; for example in c:\karaf - from now on this directory will be referenced as <KARAF_HOME>.

START THE SERVER

Open a command line console and change the directory to <KARAF_HOME>.

To start the server, run the following command in Windows:

```
bin\karaf.bat
```

respectively on Unix:

```
bin/karaf
```

You should see the following information on the command line console:

```
        __ __                  ____
       / //_/____ _____ ___  / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
```

```
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown
Karaf.

karaf@root()>
```

SOME SHELL BASICS

You can now run your first command. Simply type the <tab> key in the console.

```
karaf@root()> Display all 294 possibilities? (y or n)
...
shell:logout
shell:more
shell:new
shell:printf
shell:sleep
shell:sort                              shell:source
shell:stack-traces-print
shell:tac
shell:tail
shell:threads
shell:watch
shell:wc                                shell:while
shutdown
sleep
sort
source
ssh
ssh                                     ssh-host-change
ssh-port-change
ssh:ssh
stack-traces-print
start
start-level
status                                  stop
su
sudo
system
system:framework
system:name
system:property                         system:shutdown
system:start-level
```

```
system:version
tac
tail
threads
tree-show                           uninstall
update
user-add
user-delete
user-list
version
version-list                        wait
watch
wc                                  while
```

You can then grab more specific help for a given command using the `--help` option for this command:

```
karaf@root()> bundle:list --help
DESCRIPTION
        bundle:list

        Lists all installed bundles.

SYNTAX
        bundle:list [options] [ids]

ARGUMENTS
        ids
                The list of bundle (identified by IDs or name or
name/version) separated by whitespaces

OPTIONS
        -t
                Specifies the bundle threshold; bundles with a
start-level less than this value will not get printed out.
        --no-format
                Disable table rendered output
        -s
                Shows the symbolic name
        -l
                Show the locations
        --no-ellipsis

        --help
```

```
        Display this help message
-u
        Shows the update locations
--context, -c
        Use the given bundle context
        (defaults to 0)
-r
        Shows the bundle revisions
```

Note that the console supports tab completion so if you start typing a command it will show all possible completions and also auto complete if there is only one completion.

DEPLOY A SAMPLE APPLICATION

While you will learn in the Karaf user's guide how to fully use and leverage Apache Karaf, let's install a sample Apache Camel application for now:

In the console, run the following commands:

```
karaf@root()> feature:repo-add camel 2.15.2
Adding feature url mvn:org.apache.camel.karaf/apache-camel/
2.15.2/xml/features
karaf@root()> feature:install camel-spring
karaf@root()> bundle:install -s mvn:org.apache.camel/
camel-example-osgi/2.15.2
Bundle ID: 82
```

The example installed is using Camel to start a timer every 2 seconds and output a message in the log.
The previous commands download the Camel features descriptor and install the example feature.

You can display the log in the shell:

```
karaf@root()> log:display
...
2015-06-30 13:39:44,731 | INFO  |  timer://myTimer |
ExampleRouter                     | 53 -
org.apache.camel.camel-core - 2.15.2 | Exchange[ExchangePattern:
InOnly, BodyType: String, Body: SpringDSL set body:  Tue Jun 30
13:39:44 CEST 2015]
2015-06-30 13:39:46,730 | INFO  |  timer://myTimer |
MyTransform                       | 82 - camel-example-osgi -
2.15.2 | >>>> SpringDSL set body:  Tue Jun 30 13:39:46 CEST 2015
2015-06-30 13:39:46,731 | INFO  |  timer://myTimer |
```

```
ExampleRouter                    | 53 -
org.apache.camel.camel-core - 2.15.2 | Exchange[ExchangePattern:
InOnly, BodyType: String, Body: SpringDSL set body:  Tue Jun 30
13:39:46 CEST 2015]
2015-06-30 13:39:48,730 | INFO  |  timer://myTimer |
MyTransform                      | 82 - camel-example-osgi -
2.15.2 | >>>> SpringDSL set body:  Tue Jun 30 13:39:48 CEST 2015
2015-06-30 13:39:48,730 | INFO  |  timer://myTimer |
ExampleRouter                    | 53 -
org.apache.camel.camel-core - 2.15.2 | Exchange[ExchangePattern:
InOnly, BodyType: String, Body: SpringDSL set body:  Tue Jun 30
13:39:48 CEST 2015]
```

Stopping and uninstalling the sample application

To stop and uninstall the demo, run the following command:

```
karaf@root()> bundle:stop camel-example-osgi
karaf@root()> bundle:uninstall camel-example-osgi
```

STOPPING KARAF

To stop Karaf from the console, enter `^D` in the console:

```
^D
```

Alternatively, you can also run the following command:

```
karaf@root()> system:shutdown
```

halt is also an alias for system:shutdown:

```
karaf@root()> halt
```

Cleaning the Karaf state

Normally Karaf remembers the features and bundles you installed and started. To reset Karaf into a clean state, just delete the data directory when Karaf is not running.

SUMMARY

This document shows how simple it is to get Apache Karaf up and running and install a simple Apache Camel application.

Users Guide

Installation

Apache Karaf is a lightweight container, very easy to install and administrate, on both Unix and Windows platforms.

REQUIREMENTS

Hardware:

- 50 MB of free disk space for the Apache Karaf binary distribution.

Operating Systems:

- Windows: Windows 8, Windows 7, Windows 2003, Windows Vista, Windows XP SP2, Windows 2000.
- Unix: RedHat Enterprise Linux, Debian, SuSE/OpenSuSE, CentOS, Fedora, Ubuntu, MacOS, AIX, HP-UX, Solaris, any Unix platform that supports Java.

Environment:

- Java SE 1.7.x or greater (http://www.oracle.com/technetwork/java/javase/).
- The JAVA_HOME environment variable must be set to the directory where the Java runtime is installed,

USING APACHE KARAF BINARY DISTRIBUTIONS

Apache Karaf is available in two distributions, both as a tar.gz and zip archives.

The "default" distribution is a "ready to use" distribution.
The "default" distribution provides the following features enabled.

The "minimal" distribution is like the minimal distributions that you can find for most of Unix distributions.
Only the core layer is packaged, most of the features and bundles are downloaded from Internet at bootstrap.
It means that Apache Karaf minimal distribution requires an Internet connection to start correctly.
The features provided by the "minimal" distribution are exactly the same as in the "default" distribution, the difference
is that the minimal distribution will download the features from Internet.

Installation on Windows platform

NB: the JAVA_HOME environment variable has to be correctly defined. To accomplish that, press Windows key and Break key together, switch to "Advanced" tab and click on "Environment Variables".

1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Download Apache Karaf binary distribution in the zip format: `apache-karaf-4.0.0.zip`.
3. Extract the files from the zip file into a directory of your choice (it's the `KARAF_HOME`.
   NB: remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.
4. Apache Karaf is now installed.

Handy Hint

In case you have to install Karaf into a very deep path or a path containing illegal characters for Java paths, e.g. !, % etc., you may add a bat file to *start* \-> *startup* that executes

```
subst S: "C:\your very % problematic path!\KARAF"
```

so your Karaf root directory is S: - which works for sure and is short to type.

Installation on Unix platforms

NB: the JAVA_HOME environment variable has to be correctly defined. Check the current value using

```
echo $JAVA_HOME
```

If it's not correct, fix it using:

```
export JAVA_HOME=....
```

1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Download Apache Karaf binary distribution in the tar.gz format: `apache-karaf-4.0.0.tar.gz`.
3. Extract the files from the tar.gz file into a directory of your choice (it's the `KARAF_HOME`). For example:

   ```
   gunzip apache-karaf-4.0.0.tar.gz
   tar xvf apache-karaf-4.0.0.tar
   ```

   NB: remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.
4. Apache Karaf is now installed.

POST-INSTALLATION STEPS

Thought it is not always required, it is strongly advised to set up the `JAVA_HOME` environment property to point to the JDK you want Apache Karaf to use before starting it.
This property is used to locate the `java` executable and should be configured to point to the home directory of the Java SE 7 installation.

By default, all Apache Karaf files are "gather" in one directory: the `KARAF_HOME`.

You can define your own directory layout, by using some Karaf environment variables:

- `KARAF_DATA` is the location of the data folder, where Karaf stores temporary files.
- `KARAF_ETC` is the location of the etc folder, where Karaf stores configuration files.
- `KARAF_BASE` is the Karaf base folder. By default `KARAF_BASE` is the same as `KARAF_HOME`.

BUILDING FROM SOURCES

If you intend to build Apache Karaf from the sources, the requirements are a bit different:

Hardware:

- 500 MB of free disk space for the Apache Karaf source distributions or SVN checkout, the Maven build and the dependencies Maven downloads.

Environment:

- Java SE Development Kit 1.7.x or greater (http://www.oracle.com/technetwork/java/javase/).
- Apache Maven 3.0.4 or greater (http://maven.apache.org/download.html).

Building on Windows platform

1. You can get the Apache Karaf sources from:
- the sources distribution `apache-karaf-4.0.0-src.zip` available at http://karaf.apache.org/index/community/download.html. Extract the files in the directory of your choice.
- by checkout of the git repository:

```
git clone https://git-wip-us.apache.org/repos/asf/
karaf.git karaf
```

1. Use Apache Maven to build Apache Karaf:

```
mvn clean install
```

NB: you can speed up the build by bypassing the unit tests:

```
mvn clean install -DskipTests
```

2. You can find the built binary distribution in `assemblies\apache-karaf\target\apache-karaf-4.0.0.zip`. You can install and use it as explained in the "Using Apache Karaf binary distributions" section.

Building on Unix platforms

1. You can get the Apache Karaf sources from:

- the sources distribution `apache-karaf-4.0.0-src.tar.gz` available at http://karaf.apache.org/index/community/download.html. Extract the files in the directory of your choice.
- by checkout of the git repository:

```
git clone https://git-wip-us.apache.org/repos/asf/karaf.git karaf
```

1. Use Apache Maven to build Apache Karaf:

```
mvn clean install
```

NB: you can speed up the build by bypassing the unit tests:

```
mvn clean install -DskipTests
```

2. You can find the built binary distribution in `assemblies/apache-karaf/target/apache-karaf-4.0.0.tar.gz`. You can install and use it as explained in the "Using Apache Karaf binary distributions" section.

Directory structure

The directory layout of a Karaf installation is as follows:

- `/bin`: control scripts to start, stop, login, ...
- `/demos`: contains some simple Karaf samples
- `/etc`: configuration files
- `/data`: working directory
    - `/cache`: OSGi framework bundle cache
    - `/generated-bundles`: temporary folder used by the deployers
    - `/log`: log files
- `/deploy`: hot deploy directory
- `/instances`: directory containing instances
- `/lib`: contains libraries
    - `/lib/boot`: contains the systeù libraries used at Karaf bootstrap
    - `/lib/endorsed`: directory for endorsed libraries
    - `/lib/ext`: directory for JRE extensions
- `/system`: OSGi bundles repository, laid out as a Maven 2 repository

The `data` folder contains all the working and temporary files for Karaf.
If you want to restart from a clean state, you can wipe out this directory, which has the same effect as
using the clean option.

Start, stop, restart, connect

START

Apache Karaf supports different start mode:

- the "regular" mode starts Apache Karaf in foreground, including the shell console.
- the "server" mode starts Apache Karaf in foreground, without the shell console.
- the "background" mode starts Apache Karaf in background.

You can also manage Apache Karaf as a system service (see System Service section of this manual).

Regular mode

The regular mode uses the `bin/karaf` Unix script (`bin\karaf.bat` on Windows). It's the default start process.

It starts Apache Karaf as a foreground process, and displays the shell console.

On Unix:

```
bin/karaf
```

```
        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown
Karaf.

karaf@root()>
```

On Windows:

```
bin\karaf.bat
```

```
        __ __                  ____
       / //_/____ _____________ _/ __/
```

```
        / ,<  / __ `/ ___/ __ `/ /_
       / /| |/ /_/ / /  / /_/ / __/
      /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown
Karaf.

karaf@root()>
```

Warning
Closing the console or shell window will cause Apache Karaf to terminate.

Server mode

The server mode starts Apache Karaf as a foreground process, but it doesn't start the shell console.

To use this mode, you use the `server` argument to the `bin/karaf` Unix script (`bin\karaf.bat` on Windows).

On Unix:

```
bin/karaf server
```

On Windows:

```
bin\karaf.bat server
```

Warning
Closing the console or shell window will cause Apache Karaf to terminate.

You can connect to the shell console using SSH or client (see the Connect section in this page).

Background mode

The background mode starts Apache Karaf as a background process.

To start in background mode, you have to use `bin/start` Unix script (`bin\start.bat` on Windows).

On Unix:

```
bin/start
```

On Windows:

```
bin\start.bat
```

You can connect to the shell console using SSH or client (see the Connect section in this page).

Clean start

Apache Karaf stores all previously applications installed and changes that you did in the data folder.

If you want to start from a clean state, you can remove the data folder.

For convenience, you can use the `clean` argument to the `bin/karaf` Unix script (`bin\karaf.bat` on Windows).

On Unix:

```
bin/karaf clean
```

```
bin/start clean
```

On Windows:

```
bin\karaf.bat clean
```

```
bin\start.bat clean
```

Customize variables

Apache Karaf accepts environment variables:

- `JAVA_MIN_MEM`: minimum memory for the JVM (default is 128M).
- `JAVA_MAX_MEM`: maximum memory for the JVM (default is 512M).
- `JAVA_PERM_MEM`: minimum perm memory for the JVM (default is JVM default value).
- `JAVA_MAX_PERM_MEM`: maximum perm memory for the JVM (default is JVM default value).
- `KARAF_HOME`: the location of your Apache Karaf installation (default is found depending where you launch the startup script).

- `KARAF_BASE`: the location of your Apache Karaf base (default is `KARAF_HOME`).
- `KARAF_DATA`: the location of your Apache Karaf data folder (default is `KARAF_BASE/data`).
- `KARAF_ETC`: the location of your Apache Karaf etc folder (default is `KARAF_BASE/etc`).
- `KARAF_OPTS`: extra arguments passed to the Java command line (default is null).
- `KARAF_DEBUG`: if 'true', enable the debug mode (default is null). If debug mode is enabled, Karaf starts a JDWP socket on port 5005. You can plug your IDE to define breakpoints, and run step by step.

You can define these environment variables in `bin/setenv` Unix script (`bin\setenv.bat` on Windows).

For instance, to set the minimum and maximum memory size for the JVM, you can define the following values:

On Unix:

```
# Content of bin/setenv
export JAVA_MIN_MEM=256M
export JAVA_MAX_MEM=1024M
```

On Windows:

```
rem Content of bin\setenv.bat
set JAVA_MIN_MEM=256M
set JAVA_MAX_MEM=1024M
```

Connect

Even if you start Apache Karaf without the console (using server or background modes), you can connect to the console.
This connection can be local or remote. It means that you can access to Karaf console remotely.

To connect to the console, you can use the `bin/client` Unix script (`bin\client.bat` on Windows).

On Unix:

```
bin/client
Logging in as karaf
360 [pool-2-thread-3] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier -
Server at /0.0.0.0:8101 presented unverified key:
```

```
        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

On Windows:

```
bin\client.bat
Logging in as karaf
360 [pool-2-thread-3] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier -
Server at /0.0.0.0:8101 presented unverified key:
        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0-SNAPSHOT)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

By default, `client` tries to connect on localhost, on port 8101 (the default Apache Karaf SSH port).

`client` accepts different options to let you connect on a remote Apache Karaf instance. You can use `--help` to get details about the options:

On Unix:

```
bin/client --help
Apache Karaf client
  -a [port]     specify the port to connect to
  -h [host]     specify the host to connect to
  -u [user]     specify the user name
  --help        shows this help message
  -v            raise verbosity
  -r [attempts] retry connection establishment (up to attempts
times)
  -d [delay]    intra-retry delay (defaults to 2 seconds)
  -b            batch mode, specify multiple commands via
standard input
  -f [file]     read commands from the specified file
  [commands]    commands to run
If no commands are specified, the client will be put in an
interactive mode
```

On Windows:

```
bin\client.bat --help
Apache Karaf client
  -a [port]     specify the port to connect to
  -h [host]     specify the host to connect to
  -u [user]     specify the user name
  --help        shows this help message
  -v            raise verbosity
  -r [attempts] retry connection establishment (up to attempts
times)
  -d [delay]    intra-retry delay (defaults to 2 seconds)
  -b            batch mode, specify multiple commands via
standard input
  -f [file]     read commands from the specified file
  [commands]    commands to run
If no commands are specified, the client will be put in an
interactive mode
```

Actually, `client` is a SSH client. You can use any SSH client to connect, like OpenSSH (ssh command) on Unix, or Putty on Windows.

For instance, on Unix, you can do:

```
ssh karaf@localhost -p 8101
Authenticated with partial success.
```

```
Authenticated with partial success.
Authenticated with partial success.
Password authentication
Password:
        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0-SNAPSHOT)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

STOP

When you start Apache Karaf in regular mode, the `logout` command or CTRL-D key binding logout from the console and shutdown Apache Karaf.

When you start Apache Karaf in background mode (with the `bin/start` Unix script (`bin\start.bat` on Windows)), you can use the `bin/stop` Unix script (`bin\stop.bat` on Windows).

More generally, you can use the `shutdown` command (on the Apache Karaf console) that work in any case.

The `shutdown` command is very similar to the the `shutdown` Unix command.

To shutdown Apache Karaf now, you can simple using `shutdown`:

```
karaf@root()> shutdown -h
Confirm: halt instance root (yes/no):
```

The `shutdown` command asks for a confirmation. If you want to bypass the confirmation step, you can use the `-f` (`--force`) option:

```
karaf@root()> shutdown -f
```

You can also use directly `halt` which is an alias to `shutdown -f -h`.

The `shutdown` command accepts a time argument. With this argument, you can define when you want to shutdown the Apache Karaf container.

The time argument can have different formats. First, it can be an absolute time in the format hh:mm, in which hh is the hour (1 or 2 digits) and mm is the minute of the hour (in two digits). Second, it can be in the format m (or +m), in which m is the number of minutes to wait. The word `now` is an alias for 0.

For instance, the following command will shutdown Apache Karaf at 10:35am:

```
karaf@root()> system:shutdown 10:35
```

Another example to shutdown Apache Karaf in 10 minutes:

```
karaf@root()> system:shutdown 10
```

Like for other commands, you can find details on the `shutdown` command man page:

```
karaf@root()> shutdown --help
DESCRIPTION
        system:shutdown

        Shutdown Karaf.

SYNTAX
        system:shutdown [options] [time]

ARGUMENTS
        time
                Shutdown after a specified delay. The time
argument can have different formats. First, it can be an abolute
time in the format hh:mm, in which hh is the hour (1 or 2
digits) and mm is the minute of the hour (in two digits).
Second, it can be in the format +m, in which m is the number of
minutes to
                wait. The word now is an alias for +0.

OPTIONS
        -c, --clean, --clean-all, -ca
                Force a clean restart by deleting the data
directory
        -f, --force
                Force the shutdown without confirmation message.
        -h, --halt
                Halt the Karaf container.
```

```
        --help
                Display this help message
        -cc, --clean-cache, -cc
                Force a clean restart by deleting the cache
directory
        -r, --reboot
                Reboot the Karaf container.
```

STATUS

When you start Apache Karaf in background mode, you may want to check the current status.

To do so, you can use the `bin/status` Unix script (`bin\status.bat` on Windows).

NB: the script returns 0 exit code if Apache Karaf is running, 1 exit code else.

On Unix:

```
bin/status
Not Running ...

bin/status
Running ...
```

On Windows:

```
bin\status.bat
Not Running ...

bin\status.bat
Running ...
```

RESTART

The `shutdown` command accepts the `-r` (`--restart`) option to restart Apache Karaf:

```
karaf@root()> system:shutdown -r
```

Warning

This command does not start a new JVM. It simply restarts the OSGi framework.

SYSTEMMBEAN

Apache Karaf provides the JMX SystemMBean dedicated to control of the container itself.

The SystemMBean object name is `org.apache.karaf:type=system`.

The SystemMBean provides different attributes and operations, especially operations to halt or reboot the container:

- `reboot()` reboots Apache Karaf now (without cleaning the cache)
- `reboot(time)` reboots Apache Karaf at a given time (without cleaning the cache). The time format is the same as the time argument of the `shutdown` command.
- `rebootCleanCache(time)` reboots Apache Karaf at a given time, including the cleanup of the cache.
- `rebootCleanAll(time)` reboots Apache Karaf at a given time, including the cleanup of the whole data folder.
- `halt()` shutdown Apache Karaf now.
- `halt(time)` shutdown Apache Karaf at a given time. The time format is the same as the time argument of the `shutdown` command.

Integration in the operating system: the Service Wrapper

In the previous chapter, we saw the different scripts and commands to start, stop, restart Apache Karaf.

Instead of using these commands and scripts, you can integrate Apache Karaf directly in your operating system service control.

Apache Karaf provides the "Service Wrapper". The service wrapper allows you to directly integrate Apache Karaf:

- like a native Windows Service
- like a Unix daemon process

The "Service Wrapper" correctly handles "user's log outs" under Windows, service dependencies, and the ability to run services which interact with the desktop.

It also includes advanced fault detection software which monitors an application. The "Service Wrapper" is able to detect crashes, freezes, out of memory and other exception events, then automatically react by restarting Apache Karaf with a minimum of delay.
It guarantees the maximum possible uptime of Apache Karaf.

SUPPORTED PLATFORMS

- Windows 8, 7, 2008 R2, 2003, Vista (32 and 64 bits architecture)
- Linux RedHat Enterprise Linux, Debian, CentOS, Fedora, Ubuntu (32 and 64 bits architecture)
- FreeBSD 9.x, 8.x
- AIX 5.x, 6.x, 7.x (Power architecture)
- Solaris 8, 9, 10 (x86/Sparc, 32 and 64 bits architecture)
- HP-UX 10.x, 11.x (32 and 64 bits architecture)
- SGI Irix
- MacOS X

INSTALLATION

Apache Karaf Service Wrapper is an optional feature. You have to install the "Service Wrapper" installer first.

In the console:

```
karaf@root()> feature:install service-wrapper
```

Now, you have the `wrapper:install` command, to "register" Apache Karaf as service/daemon on your system:

```
karaf@root()> wrapper:install --help
DESCRIPTION
        wrapper:install

        Install the container as a system service in the OS.

SYNTAX
        wrapper:install [options]

OPTIONS
        -d, --display
                The display name of the service.
                (defaults to karaf)
        --help
                Display this help message
        -s, --start-type
                Mode in which the service is installed.
AUTO_START or DEMAND_START (Default: AUTO_START)
                (defaults to AUTO_START)
        -n, --name
                The service name that will be used when
installing the service. (Default: karaf)
                (defaults to karaf)
        -D, --description
                The description of the service.
                (defaults to )
```

INSTALLATION

Karaf Wrapper is an optional feature. To install it, simply type:

```
karaf@root> feature:install wrapper
```

Once installed, wrapper feature will provide `wrapper:install` new command in the Karaf shell:

```
karaf@root> wrapper:install --help
DESCRIPTION
```

```
        wrapper:install

        Install the container as a system service in the OS.

SYNTAX
        wrapper:install [options]

OPTIONS
        -s, --start-type
                Mode in which the service is installed.
AUTO_START or DEMAND_START (Default: AUTO_START)
                (defaults to AUTO_START)
        --help
                Display this help message
        -n, --name
                The service name that will be used when
installing the service. (Default: karaf)
                (defaults to karaf)
        -d, --display
                The display name of the service.
        -D, --description
                The description of the service.
                (defaults to )
```

The `wrapper:install` command detects the running Operating Service and provide the service/daemon ready to be integrated in your system.

For instance, on a Ubuntu/Debian Linux system:

```
karaf@root()> wrapper:install
Creating file: /opt/apache-karaf-4.0.0/bin/karaf-wrapper
Creating file: /opt/apache-karaf-4.0.0/bin/karaf-service
Creating file: /opt/apache-karaf-4.0.0/etc/karaf-wrapper.conf
Creating missing directory: /opt/apache-karaf-4.0.0/lib/wrapper
Creating file: /opt/apache-karaf-4.0.0/lib/wrapper/libwrapper.so
Creating file: /opt/apache-karaf-4.0.0/lib/wrapper/
karaf-wrapper.jar
Creating file: /opt/apache-karaf-4.0.0/lib/wrapper/
karaf-wrapper-main.jar

Setup complete.  You may wish to tweak the JVM properties in the
wrapper configuration file:
        /opt/apache-karaf-4.0.0/etc/karaf-wrapper.conf
before installing and starting the service.
```

```
Ubuntu/Debian Linux system detected:
  To install the service:
    $ ln -s /opt/apache-karaf-4.0.0/bin/karaf-service /etc/
init.d/

  To start the service when the machine is rebooted:
    $ update-rc.d karaf-service defaults

  To disable starting the service when the machine is rebooted:
    $ update-rc.d -f karaf-service remove

  To start the service:
    $ /etc/init.d/karaf-service start

  To stop the service:
    $ /etc/init.d/karaf-service stop

  To uninstall the service :
    $ rm /etc/init.d/karaf-service
```

You can note that `wrapper:install` command detected the running operating system ("Ubuntu/Debian Linux system detected").

You have a complete explanation and list of system commands to perform to integrate Apache Karaf in your systemV:

```
ln -s /opt/apache-karaf-4.0.0/bin/karaf-service /etc/init.d/
update-rc.d karaf-service defaults
```

Karaf also supports systemd service, so you can use systemctl instead of SystemV based service:

```
systemctl enable /opt/apache-karaf-4.0.2/bin/karaf.service
```

This will enable Karaf at system boot.

UNINSTALL

The `wrapper:install` provides the system commands to perform to uninstall the service/daemon).

For instance, on Ubuntu/Debian, to uninstall the Apache Karaf service, you have to remove the `karaf-service` script from the runlevel scripts:

```
rm /etc/init.d/karaf-service
```

If you prefered the systemd service instead of systemV:

```
systemctl disable karaf
```

You can remove the "Wrapper Service" installer after that:

```
karaf@root()> feature:uninstall service-wrapper
```

NOTE FOR MACOS USERS

On MacOS you can install the service for an user or for the system.

If you want to add bin/org.apache.karaf.KARAF as user service move this file into ~/Library/LaunchAgents/:

```
mv bin/org.apache.karaf.KARAF.plist ~/Library/LaunchAgents/
```

If you want to add org.apache.karaf.KARAF as system service move this into /Library/LaunchDaemons:

```
sudo mv bin/org.apache.karaf.KARAF.plist /Library/LaunchDaemons/
```

Change owner and rights:

```
sudo chown root:wheel /Library/LaunchDaemons/
org.apache.karaf.KARAF.plist
sudo chmod u=rw,g=r,o=r /Library/LaunchDaemons/
org.apache.karaf.KARAF.plist
```

You can test your service:

```
launchctl load ~/Library/LaunchAgents/
org.apache.karaf.KARAF.plist
launchctl start org.apache.karaf.KARAF
launchctl stop org.apache.karaf.KARAF
```

Finally, after restart your session or system you can use launchctl command to start and stop your service.

If you want to remove the service call:

```
launchctl remove org.apache.karaf.KARAF
```

CONFIGURATION

When using scripts in the Apache Karaf `bin` folder, you can using `bin/setenv` Unix script (`bin\setenv.bat` on Windows) as described in the Start, stop, restart, connect section of the documentation.

> **Warning**
>
> The `bin/setenv` Unix script (`bin\setenv.bat` on Windows) is not used by the Apache Karaf Service Wrapper.

To configure Apache Karaf started by the Service Wrapper, you have to tune the `etc/karaf-wrapper.conf` file. If you provided the `name` option to the `wrapper:install` command, the file is `etc/karaf-yourname.conf`.

In this file, you can configure the different environment variables used by Apache Karaf. The Service Wrapper installer automatically populate these variables for you during the installation (using `wrapper:install` command).
For instance:

- `set.default.JAVA_HOME` is the `JAVA_HOME` used to start Apache Karaf (populated during Service Wrapper installation).
- `set.default.KARAF_HOME` is the location of your Apache Karaf installation (populated during Service Wrapper installation).
- `set.default.KARAF_BASE` is the location of your Apache Karaf installation (populated during Service Wrapper installation).
- `set.default.KARAF_DATA` is the location of the Apache Karaf data folder (populated during Service Wrapper installation).
- `set.default.KARAF_ETC` is the location of the Apache Karaf etc folder (populated during Service Wrapper installation).
- `wrapper.java.additional` is used to pass additional arguments to the Java command, indexed by the argument number. The next index to use is 11.
- `wrapper.java.initmemory` is the initial JVM memory size (the `-Xms`). It's not set by default (JVM default).
- `wrapper.java.maxmemory` is the maximum JVM memory size (the `-Xmx`). It's set to 512M by default.
- `wrapper.logfile` is the location of the Service Wrapper log file. It's set to `%KARAF_DATA%/log/wrapper.log` by default.
- `wrapper.logfile.loglevel` is the Service Wrapper log level. It's set to `INFO` by default.
- `wrapper.logfile.maxsize` is the Service Wrapper log file maximum size (before rotation). It's set to `10m` (10MB) by default.

- `wrapper.logfile.maxfiles` is the number of Service Wrapper log files created (and rotated). It's set to `5` by default.
- `wrapper.syslog.loglevel` is the integration with Unix syslog daemon. By default, it's set to `none` meaning disabled.
- `wrapper.ntservice.name` is Windows service specific and defines the Windows service name. It's set to the `name` option of the `wrapper:install` command, or `karaf` by default.
- `wrapper.ntservice.displayname` is Windows service specific and defines the Windows service display name. It's set to the `display` option of the `wrapper:install` command, or `karaf` by default.
- `wrapper.ntservice.description` is Windows service specific and defines the Windows service description. It's set to the `description` option of the `wrapper:install` command, or empty by default.
- `wrapper.ntservice.starttype` is Windows service specific and defines if the Windows service is started automatically with the service, or just on demand. It's set to `AUTO_START` by default, and could be switch to `DEMAND_START`.

This is a example of generated `etc/karaf-wrapper.conf` file:

```
#
#---------------------------------------------------------------------------
# Licensed to the Apache Software Foundation (ASF) under one or
more
# contributor license agreements.  See the NOTICE file
distributed with
# this work for additional information regarding copyright
ownership.
# The ASF licenses this file to You under the Apache License,
Version 2.0
# (the "License"); you may not use this file except in
compliance with
# the License.  You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS"
BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
# See the License for the specific language governing
```

```
permissions and
# limitations under the License.
#
-------------------------------------------------------------------------

#*********************************************************************
# Wrapper Properties
#*********************************************************************
set.default.JAVA_HOME=/opt/jdk/1.7.0_21
set.default.KARAF_HOME=/opt/apache-karaf-4.0.0
set.default.KARAF_BASE=/opt/apache-karaf-4.0.0
set.default.KARAF_DATA=/opt/apache-karaf-4.0.0/data
set.default.KARAF_ETC=/opt/apache-karaf-4.0.0/etc

# Java Application
wrapper.working.dir=%KARAF_BASE%
wrapper.java.command=%JAVA_HOME%/bin/java
wrapper.java.mainclass=org.apache.karaf.wrapper.internal.Main
wrapper.java.classpath.1=%KARAF_HOME%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_HOME%/lib/karaf-wrapper-main.jar
wrapper.java.classpath.5=%KARAF_HOME%/lib/karaf-org.osgi.core.jar
wrapper.java.library.path.1=%KARAF_HOME%/lib/

# Application Parameters.  Add parameters as needed starting
from 1
#wrapper.app.parameter.1=

# JVM Parameters
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dkaraf.home=%KARAF_HOME%
wrapper.java.additional.2=-Dkaraf.base=%KARAF_BASE%
wrapper.java.additional.3=-Dkaraf.data=%KARAF_DATA%
wrapper.java.additional.4=-Dkaraf.etc=%KARAF_ETC%
wrapper.java.additional.5=-Dcom.sun.management.jmxremote
wrapper.java.additional.6=-Dkaraf.startLocalConsole=false
wrapper.java.additional.7=-Dkaraf.startRemoteShell=true
wrapper.java.additional.8=-Djava.endorsed.dirs=%JAVA_HOME%/jre/
lib/endorsed:%JAVA_HOME%/lib/endorsed:%KARAF_HOME%/lib/endorsed
wrapper.java.additional.9=-Djava.ext.dirs=%JAVA_HOME%/jre/lib/
ext:%JAVA_HOME%/lib/ext:%KARAF_HOME%/lib/ext
```

```
# Uncomment to enable jmx
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.port=1616
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.authenticate=fals
#wrapper.java.additional.n=-Dcom.sun.management.jmxremote.ssl=false

# Uncomment to enable YourKit profiling
#wrapper.java.additional.n=-Xrunyjpagent

# Uncomment to enable remote debugging
#wrapper.java.additional.n=-Xdebug -Xnoagent -Djava.compiler=NONE
#wrapper.java.additional.n=-Xrunjdwp:transport=dt_socket,server=y,suspend=n

# Initial Java Heap Size (in MB)
#wrapper.java.initmemory=3

# Maximum Java Heap Size (in MB)
wrapper.java.maxmemory=512


#*********************************************************************
# Wrapper Logging Properties
#*********************************************************************
# Format of output for the console.  (See docs for formats)
wrapper.console.format=PM

# Log Level for console output.  (See docs for log levels)
wrapper.console.loglevel=INFO

# Log file to use for wrapper output logging.
wrapper.logfile=%KARAF_DATA%/log/wrapper.log

# Format of output for the log file.  (See docs for formats)
wrapper.logfile.format=LPTM

# Log Level for log file output.  (See docs for log levels)
wrapper.logfile.loglevel=INFO

# Maximum size that the log file will be allowed to grow to
before
#  the log is rolled. Size is specified in bytes.  The default
value
#  of 0, disables log rolling.  May abbreviate with the 'k' (kb)
or
```

```
#  'm' (mb) suffix.  For example: 10m = 10 megabytes.
wrapper.logfile.maxsize=10m

# Maximum number of rolled log files which will be allowed
before old
#  files are deleted.  The default value of 0 implies no limit.
wrapper.logfile.maxfiles=5

# Log Level for sys/event log output.  (See docs for log levels)
wrapper.syslog.loglevel=NONE

#********************************************************************
# Wrapper Windows Properties
#********************************************************************
# Title to use when running as a console
wrapper.console.title=karaf

#********************************************************************
# Wrapper Windows NT/2000/XP Service Properties
#********************************************************************
# WARNING - Do not modify any of these properties when an
application
#  using this configuration file has been installed as a service.
#  Please uninstall the service before modifying this section.
The
#  service can then be reinstalled.

# Name of the service
wrapper.ntservice.name=karaf

# Display name of the service
wrapper.ntservice.displayname=karaf

# Description of the service
wrapper.ntservice.description=

# Service dependencies.  Add dependencies as needed starting
from 1
wrapper.ntservice.dependency.1=

# Mode in which the service is installed.  AUTO_START or
DEMAND_START
wrapper.ntservice.starttype=AUTO_START
```

```
# Allow the service to interact with the desktop.
wrapper.ntservice.interactive=false
```

systemd

The Karaf service wrapper also support Linux systemd service.

Using the console

AVAILABLE COMMANDS

To see a list of the available commands in the console, you can use the `help`:

```
karaf@root()> help
bundle                              Enter the subshell
bundle:capabilities                 Displays OSGi capabilities of
a given bundles.
bundle:classes                      Displays a list of classes/
resources contained in the bundle
bundle:diag                         Displays diagnostic
information why a bundle is not Active
bundle:dynamic-import               Enables/disables
dynamic-import for a given bundle.
bundle:find-class                   Locates a specified class in
any deployed bundle
bundle:headers                      Displays OSGi headers of a
given bundles.
bundle:id                           Gets the bundle ID.
...
```

You have the list of all commands with a short description.

You can use the tab key to get a quick list of all commands:

```
karaf@root()> Display all 294 possibilities? (y or n)
...
```

SUBSHELL AND COMPLETION MODE

The commands have a scope and a name. For instance, the command `feature:list` has `feature` as scope, and `list` as name.

Karaf "groups" the commands by scope. Each scope form a subshell.

You can directly execute a command with its full qualified name (scope:name):

```
karaf@root()> feature:list
...
```

or enter in a subshell and type the command contextual to the subshell:

```
karaf@root()> feature
karaf@root(feature)> list
```

You can note that you enter in a subshell directly by typing the subshell name (here `feature`). You can "switch" directly from a subshell to another:

```
karaf@root()> feature
karaf@root(feature)> bundle
karaf@root(bundle)>
```

The prompt displays the current subshell between ().

The `exit` command goes to the parent subshell:

```
karaf@root()> feature
karaf@root(feature)> exit
karaf@root()>
```

The completion mode defines the behaviour of the tab key and the help command.

You have three different modes available:

- GLOBAL
- FIRST
- SUBSHELL

You can define your default completion mode using the completionMode property in `etc/org.apache.karaf.shell.cfg` file. By default, you have:

```
completionMode = GLOBAL
```

You can also change the completion mode “on the fly” (while using the Karaf shell console) using the `shell:completion` command:

```
karaf@root()> shell:completion
GLOBAL
karaf@root()> shell:completion FIRST
karaf@root()> shell:completion
FIRST
```

`shell:completion` can inform you about the current completion mode used. You can also provide the new completion mode that you want.

GLOBAL completion mode is the default one in Karaf 4.0.0 (mostly for transition purpose).

GLOBAL mode doesn’t really use subshell: it’s the same behavior as in previous Karaf versions.

When you type the tab key, whatever in which subshell you are, the completion will display all commands and all aliases:

```
karaf@root()> <TAB>
karaf@root()> Display all 273 possibilities? (y or n)
...
karaf@root()> feature
karaf@root(feature)> <TAB>
karaf@root(feature)> Display all 273 possibilities? (y or n)
```

FIRST completion mode is an alternative to the GLOBAL completion mode.

If you type the tab key on the root level subshell, the completion will display the commands and the aliases from all subshells (as in GLOBAL mode).
However, if you type the tab key when you are in a subshell, the completion will display only the commands of the current subshell:

```
karaf@root()> shell:completion FIRST
karaf@root()> <TAB>
karaf@root()> Display all 273 possibilities? (y or n)
...
karaf@root()> feature
karaf@root(feature)> <TAB>
karaf@root(feature)>
info install list repo-add repo-list repo-remove uninstall
version-list
karaf@root(feature)> exit
karaf@root()> log
karaf@root(log)> <TAB>
karaf@root(log)>
clear display exception-display get log set tail
```

SUBSHELL completion mode is the real subshell mode.

If you type the tab key on the root level, the completion displays the subshell commands (to go into a subshell), and the global aliases.
Once you are in a subshell, if you type the TAB key, the completion displays the commands of the current subshell:

```
karaf@root()> shell:completion SUBSHELL
karaf@root()> <TAB>
karaf@root()>
* bundle cl config dev feature help instance jaas kar la ld lde
log log:list man package region service shell ssh system
karaf@root()> bundle
karaf@root(bundle)> <TAB>
```

```
karaf@root(bundle)>
capabilities classes diag dynamic-import find-class headers info
install list refresh requirements resolve restart services start
start-level stop
uninstall update watch
karaf@root(bundle)> exit
karaf@root()> camel
karaf@root(camel)> <TAB>
karaf@root(camel)>
backlog-tracer-dump backlog-tracer-info backlog-tracer-start
backlog-tracer-stop context-info context-list context-start
context-stop endpoint-list route-info route-list route-profile
route-reset-stats
route-resume route-show route-start route-stop route-suspend
```

UNIX LIKE ENVIRONMENT

Karaf console provides a full Unix like environment.

Help or man

We already saw the usage of the `help` command to display all commands available.

But you can also use the `help` command to get details about a command or the `man` command which is an alias to the `help` command.
You can also use another form to get the command help, by using the `--help` option to the command.

So these commands

```
karaf@root()> help feature:list
karaf@root()> man feature:list
karaf@root()> feature:list --help
```

All produce the same help output:

```
DESCRIPTION
        feature:list

        Lists all existing features available from the defined
repositories.

SYNTAX
        feature:list [options]
```

```
OPTIONS
        --help
                Display this help message
        -o, --ordered
                Display a list using alphabetical order
        -i, --installed
                Display a list of all installed features only
        --no-format
                Disable table rendered output
```

Completion

When you type the tab key, Karaf tries to complete:

- subshell
- commands
- aliases
- command arguments
- command options

Alias

An alias is another name associated to a given command.

The `shell:alias` command creates a new alias. For instance, to create the `list-installed-features` alias to the actual `feature:list -i` command, you can do:

```
karaf@root()> alias "list-features-installed = { feature:list -i
}"
karaf@root()> list-features-installed
Name        | Version | Required | State   | Repository     |
Description
--------------------------------------------------------------------------------
feature     | 4.0.0   | x        | Started | standard-4.0.0 |
Features Support
shell       | 4.0.0   | x        | Started | standard-4.0.0 |
Karaf Shell
deployer    | 4.0.0   | x        | Started | standard-4.0.0 |
Karaf Deployer
bundle      | 4.0.0   | x        | Started | standard-4.0.0 |
Provide Bundle support
config      | 4.0.0   | x        | Started | standard-4.0.0 |
Provide OSGi ConfigAdmin support
```

```
diagnostic | 4.0.0   | x        | Started | standard-4.0.0 |
Provide Diagnostic support
instance   | 4.0.0   | x        | Started | standard-4.0.0 |
Provide Instance support
jaas       | 4.0.0   | x        | Started | standard-4.0.0 |
Provide JAAS support
log        | 4.0.0   | x        | Started | standard-4.0.0 |
Provide Log support
package    | 4.0.0   | x        | Started | standard-4.0.0 |
Package commands and mbeans
service    | 4.0.0   | x        | Started | standard-4.0.0 |
Provide Service support
system     | 4.0.0   | x        | Started | standard-4.0.0 |
Provide System support
kar        | 4.0.0   | x        | Started | standard-4.0.0 |
Provide KAR (KARaf archive) support
ssh        | 4.0.0   | x        | Started | standard-4.0.0 |
Provide a SSHd server on Karaf
management | 4.0.0   | x        | Started | standard-4.0.0 |
Provide a JMX MBeanServer and a set of MBeans in
```

At login, the Apache Karaf console reads the `etc/shell.init.script` file where you can create your aliases.
It's similar to a bashrc or profile file on Unix.

```
ld = { log:display $args } ;
lde = { log:exception-display $args } ;
la = { bundle:list -t 0 $args } ;
ls = { service:list $args } ;
cl = { config:list "(service.pid=$args)" } ;
halt = { system:shutdown -h -f $args } ;
help = { *:help $args | more } ;
man = { help $args } ;
log:list = { log:get ALL } ;
```

You can see here the aliases available by default:

- `ld` is a short form to display log (alias to `log:display` command)
- `lde` is a short form to display exceptions (alias to `log:exception-display` command)
- `la` is a short form to list all bundles (alias to `bundle:list -t 0` command)
- `ls` is a short form to list all services (alias to `service:list` command)
- `cl` is a short form to list all configurations (alias to `config:list` command)

- `halt` is a short form to shutdown Apache Karaf (alias to `system:shutdown -h -f` command)
- `help` is a short form to display help (alias to `*:help` command)
- `man` is the same as help (alias to `help` command)
- `log:list` displays all loggers and level (alias to `log:get ALL` command)

You can create your own aliases in the `etc/shell.init.script` file.

Key binding

Like on most Unix environment, Karaf console support some key bindings:

- the arrows key to navigate in the commands history
- CTRL-D to logout/shutdown Karaf
- CTRL-R to search previously executed command
- CTRL-U to remove the current line

Pipe

You can pipe the output of one command as input to another one. It's a pipe, using the | character:

```
karaf@root()> feature:list |grep -i war
pax-war                         | 4.1.4
|          | Uninstalled | org.ops4j.pax.web-4.1.4  | Provide
support of a full WebContainer
pax-war-tomcat                  | 4.1.4
|          | Uninstalled | org.ops4j.pax.web-4.1.4  |
war                             | 4.0.0
|          | Uninstalled | standard-4.0.0           | Turn Karaf
as a full WebContainer
blueprint-web                   | 4.0.0
|          | Uninstalled | standard-4.0.0           | Provides
an OSGI-aware Servlet ContextListener fo
```

Grep, more, find, ...

Karaf console provides some core commands similar to Unix environment:

- `shell:alias` creates an alias to an existing command
- `shell:cat` displays the content of a file or URL
- `shell:clear` clears the current console display
- `shell:completion` displays or change the current completion mode
- `shell:date` displays the current date (optionally using a format)
- `shell:each` executes a closure on a list of arguments
- `shell:echo` echoes and prints arguments to stdout

- `shell:edit` calls a text editor on the current file or URL
- `shell:env` displays or sets the value of a shell session variable
- `shell:exec` executes a system command
- `shell:grep` prints lines matching the given pattern
- `shell:head` displays the first line of the input
- `shell:history` prints the commands history
- `shell:if` allows you to use conditions (if, then, else blocks) in script
- `shell:info` prints various information about the current Karaf instance
- `shell:java` executes a Java application
- `shell:less` file pager
- `shell:logout` disconnects shell from current session
- `shell:more` is a file pager
- `shell:new` creates a new Java object
- `shell:printf` formats and prints arguments
- `shell:sleep` sleeps for a bit then wakes up
- `shell:sort` writes sorted concatenation of all files to stdout
- `shell:source` executes commands contained in a script
- `shell:stack-traces-print` prints the full stack trace in the console when the execution of a command throws an exception
- `shell:tac` captures the STDIN and returns it as a string
- `shell:tail` displays the last lines of the input
- `shell:threads` prints the current thread
- `shell:watch` periodically executes a command and refresh the output
- `shell:wc` prints newline, words, and byte counts for each file
- `shell:while` loop while the condition is true

You don't have to use the fully qualified name of the command, you can directly use the command name as long as it is unique.
So you can use 'head' instead of 'shell:head'

Again, you can find details and all options of these commands using `help` command or `--help` option.

Scripting

The Apache Karaf Console supports a complete scripting language, similar to bash or csh on Unix.

The `each` (`shell:each`) command can iterate in a list:

```
karaf@root()> list = [1 2 3]; each ($list) { echo $it }
1
2
3
```

The same loop could be written with the `shell:while` command:

```
karaf@root()> a = 0 ; while { %((a+=1) <= 3) } { echo $a
}
1
2
3
```

You can create the list yourself (as in the previous example), or some commands can return a list too.

We can note that the console created a "session" variable with the name `list` that you can access with `$list`.

The `$it` variable is an implicit one corresponding to the current object (here the current iterated value from the
list).

When you create a list with `[]`, Apache Karaf console creates a Java ArrayList. It means that you can use methods
available in the ArrayList objects (like get or size for instance):

```
karaf@root()> list = ["Hello" world]; echo ($list get 0) ($list
get 1)
Hello world
```

We can note here that calling a method on an object is directly using `(object method argument)`.
Here `($list get 0)` means `$list.get(0)` where `$list` is the ArrayList.

The `class` notation will display details about the object:

```
karaf@root()> $list class
...
ProtectionDomain     ProtectionDomain  null
 null
 <no principals>
 java.security.Permissions@6521c24e (
 ("java.security.AllPermission" "<all permissions>" "<all
actions>")
)


Signers              null
```

```
SimpleName            ArrayList
TypeParameters        [E]
```

You can "cast" a variable to a given type.

```
karaf@root()> ("hello world" toCharArray)
[h, e, l, l, o,  , w, o, r, l, d]
```

If it fails, you will see the casting exception:

```
karaf@root()> ("hello world" toCharArray)[0]
Error executing command: [C cannot be cast to [Ljava.lang.Object;
```

You can "call" a script using the `shell:source` command:

```
karaf@root> shell:source script.txt
True!
```

where `script.txt` contains:

```
foo = "foo"
if { $foo equals "foo" } {
  echo "True!"
}
```

> The spaces are important when writing script.
> For instance, the following script is not correct:
>
> ```
> if{ $foo equals "foo" } ...
> ```
>
> and will fail with:
>
> ```
> karaf@root> shell:source script.txt
> Error executing command: Cannot coerce echo "true!"() to
> any of []
> ```
>
> because a space is missing after the `if` statement.

As for the aliases, you can create init scripts in the `etc/shell.init.script` file.
You can also named you script with an alias. Actually, the aliases are just scripts.

See the Scripting section of the developers guide for details.

SECURITY

The Apache Karaf console supports a Role Based Access Control (RBAC) security mechanism. It means that depending of
the user connected to the console, you can define, depending of the user's groups and roles, the permission to execute
some commands, or limit the values allowed for the arguments.

Console security is detailed in the Security section of this user guide.

Remote

Apache Karaf supports a complete remote mechanism allowing you to remotely connect to a running Apache Karaf instance.
More over, you can also browse, download, and upload files remotely to a running Apache Karaf instance.

Apache Karaf embeds a complete SSHd server.

SSHD SERVER

When you start Apache Karaf, it enables a remote console that can be accessed over SSH.

This remote console provides all the features of the "local" console, and gives a remote user complete control over the
container and services running inside of it. As the "local" console, the remote console is secured by a RBAC mechanism
(see the Security section of the user guide for details).

In addition of the remote console, Apache Karaf also provides a remote filesystem. This remote filesystem can be accessed
using a SCP/SFTP client.

Configuration

The configuration of the SSHd server is stored in the `etc/org.apache.karaf.shell.cfg` file:

```
################################################################################
#
#    Licensed to the Apache Software Foundation (ASF) under one
or more
#    contributor license agreements.  See the NOTICE file
distributed with
#    this work for additional information regarding copyright
ownership.
#    The ASF licenses this file to You under the Apache License,
Version 2.0
#    (the "License"); you may not use this file except in
compliance with
#    the License.  You may obtain a copy of the License at
```

```
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing,
software
#    distributed under the License is distributed on an "AS IS"
BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied.
#    See the License for the specific language governing
permissions and
#    limitations under the License.
#
################################################################################

#
# These properties are used to configure Karaf's ssh shell.
#

#
# Via sshPort and sshHost you define the address you can login
into Karaf.
#
sshPort = 8101
sshHost = 0.0.0.0

#
# The sshIdleTimeout defines the inactivity timeout to logout
the SSH session.
# The sshIdleTimeout is in milliseconds, and the default is set
to 30 minutes.
#
sshIdleTimeout = 1800000

#
# sshRealm defines which JAAS domain to use for password
authentication.
#
sshRealm = karaf

#
# The location of the hostKey file defines where the private/
public key of the server
```

```
# is located. If no file is at the defined location it will be
ignored.
#
hostKey = ${karaf.etc}/host.key

#
# Role name used for SSH access authorization
# If not set, this defaults to the ${karaf.admin.role}
configured in etc/system.properties
#
# sshRole = admin

#
# Self defined key size in 1024, 2048, 3072, or 4096
# If not set, this defaults to 4096.
#
# keySize = 4096

#
# Specify host key algorithm, defaults to RSA
#
# algorithm = RSA

#
# Defines the completion mode on the Karaf shell console. The
possible values are:
# - GLOBAL: it's the same behavior as in previous Karaf
releases. The completion displays all commands and all aliases
#           ignoring if you are in a subshell or not.
# - FIRST: the completion displays all commands and all aliases
only when you are not in a subshell. When you are
#          in a subshell, the completion displays only the
commands local to the subshell.
# - SUBSHELL: the completion displays only the subshells on the
root level. When you are in a subshell, the completion
#             displays only the commands local to the subshell.
# This property define the default value when you use the Karaf
shell console.
# You can change the completion mode directly in the shell
console, using shell:completion command.
#
completionMode = GLOBAL
```

The `etc/org.apache.karaf.shell.cfg` configuration file contains different properties to configure the SSHd server:

- `sshPort` is the port number where the SSHd server is bound (by default, it's 8101).
- `sshHost` is the address of the network interface where the SSHd server is bound. The default value is 0.0.0.0,
  meaning that the SSHd server is bound on all network interfaces. You can bind on a target interface providing the IP
  address of the network interface.
- `hostKey` is the location of the `host.key` file. By defaut, it uses `etc/host.key`. This file stores the public
  and private key pair of the SSHd server.
- `sshRole` is the default role used for SSH access. The default value is the value of `karaf.admin.role` property
  defined in `etc/system.properties`. See the Security section of this user guide for details.
- `keySize` is the key size used by the SSHd server. The possible values are 1024, 2048, 3072, or 4096. The default
  value is 1024.
- `algorithm` is the host key algorithm used by the SSHd server. The possible values are DSA or RSA. The default
  value is DSA.

The SSHd server configuration can be changed at runtime:

- by editing the `etc/org.apache.karaf.shell.cfg` configuration file
- by using the `config:*` commands

At runtime, when you change the SSHd server configuration, you have to restart the SSHd server to load the changes.
You can do it with:

```
karaf@root()> bundle:restart -f org.apache.karaf.shell.ssh
```

The Apache Karaf SSHd server supports key/agent authentication and password authentication.

Console clients

System native clients

The Apache Karaf SSHd server is a pure SSHd server, similar to OpenSSH daemon.

It means that you can use directly a SSH client from your system.

For instance, on Unix, you can directly use OpenSSH:

```
~$ ssh -p 8101 karaf@localhost
Authenticated with partial success.
Authenticated with partial success.
Authenticated with partial success.
Password authentication
Password:

        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

On Windows, you can use Putty, Kitty, etc.

If you don't have SSH client installed on your machine, you can use Apache Karaf client.

`ssh:ssh` command

Apache Karaf itself provides a SSH client. When you are on the Apache Karaf console, you have the `ssh:ssh` command:

```
karaf@root()> ssh:ssh --help
DESCRIPTION
        ssh:ssh

        Connects to a remote SSH server

SYNTAX
        ssh:ssh [options] hostname [command]

ARGUMENTS
        hostname
                The host name to connect to via SSH
```

```
        command
                Optional command to execute

OPTIONS
        --help
                Display this help message
        -p, --port
                The port to use for SSH connection
                (defaults to 22)
        -P, --password
                The password for remote login
        -q
                Quiet Mode. Do not ask for confirmations
        -l, --username
                The user name for remote login
```

Thanks to the `ssh:ssh` command, you can connect to another running Apache Karaf instance:

```
karaf@root()> ssh:ssh -p 8101 karaf@192.168.134.2
Connecting to host 192.168.134.2 on port 8101
Connecting to unknown server. Add this server to known hosts ?
(y/n)
Storing the server key in known_hosts.
Connected

        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

When you don't provide the `command` argument to the `ssh:ssh` command, you are in the interactive mode: you have
a complete remote console available, where you can type commands, etc.

You can also provide directly a command to execute using the `command` argument. For instance, to remotely shutdown
a Apache Karaf instance:

```
karaf@root()> ssh:ssh -p 8101 karaf@localhost system:shutdown -f
Connecting to host localhost on port 8101
Connected
```

As the `ssh:ssh` command is a pure SSH client, so it means that you can connect to a Unix OpenSSH daemon:

```
karaf@root()> ssh:ssh user@localhost
Connecting to host localhost on port 22
Connecting to unknown server. Add this server to known hosts ?
(y/n)
Storing the server key in known_hosts.
Agent authentication failed, falling back to password
authentication.
Password: Connected
Last login: Sun Sep  8 19:21:12 2013
user@server:~$
```

Apache Karaf client

The `ssh:ssh` command requires to be run into a running Apache Karaf console.

For commodity, the `ssh:ssh` command is "wrapped" as a standalone client: the `bin/client` Unix script (`bin\client.bat` on Windows).

```
bin/client --help
Apache Karaf client
  -a [port]     specify the port to connect to
  -h [host]     specify the host to connect to
  -u [user]     specify the user name
  --help        shows this help message
  -v            raise verbosity
  -r [attempts] retry connection establishment (up to attempts
times)
  -d [delay]    intra-retry delay (defaults to 2 seconds)
  -b            batch mode, specify multiple commands via
standard input
```

```
  -f [file]     read commands from the specified file
  [commands]    commands to run
If no commands are specified, the client will be put in an
interactive mode
```

For instance, to connect to local Apache Karaf instance (on the default SSHd server 8101 port), you can directly use
`bin/client` Unix script (`bin\client.bat` on Windows) without any argument or option:

```
bin/client
Logging in as karaf
343 [pool-2-thread-4] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier -
Server at /0.0.0.0:8101 presented unverified key:

        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current
session.

karaf@root()>
```

When you don't provide the `command` argument to the `bin/client` Unix script (`bin\client.bat` on Windows), you are
in the interactive mode: you have a complete remote console available, where you can type commands, etc.

You can also provide directly a command to execute using the `command` argument. For instance, to remotely shutdown
a Apache Karaf instance:

```
bin/client "system:shutdown -f"
Logging in as karaf
330 [pool-2-thread-3] WARN
```

```
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier -
Server at /0.0.0.0:8101 presented unverified key:
```

As the Apache Karaf client is a pure SSH client, you can use to connect to any SSHd daemon (like Unix OpenSSH daemon):

```
bin/client -a 22 -h localhost -u user
Logging in as user
353 [pool-2-thread-2] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier -
Server at localhost/127.0.0.1:22 presented unverified key:
Password:
Welcome to Ubuntu 13.10 (GNU/Linux 3.11.0-13-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

Last login: Tue Dec  3 18:18:31 2013 from localhost
```

Logout

When you are connected to a remote Apache Karaf console, you can logout using:

- using CTRL-D key binding. Note that CTRL-D just logout from the remote console in this case, it doesn't shutdown
  the Apache Karaf instance (as CTRL-D does when used on a local console).
- using `shell:logout` command (or simply `logout`)

Filsystem clients

Apache Karaf SSHd server also provides complete fileystem access via SSH. For security reason, the available filesystem
is limited to `KARAF_BASE` directory.

You can use this remote filesystem with any SCP/SFTP compliant clients.

Native SCP/SFTP clients

On Unix, you can directly use `scp` command to download/upload files to the Apache Karaf filesystem. For instance,
to retrieve the `karaf.log` file remotely:

```
~$ scp -P 8101 karaf@localhost:/data/log/karaf.log .
Authenticated with partial success.
Authenticated with partial success.
```

```
Authenticated with partial success.
Password authentication
Password:
karaf.log
```

As you have access to the complete `KARAF_BASE` directory, you can remotely change the configuration file in the `etc`
folder, retrieve log files, populate the `system` folder.

On Windows, you can use WinSCP to access the Apache Karaf filesystem.

It's probably easier to use a SFTP complient client.

For instance, on Unix system, you can use `lftp` or `ncftp`:

```
$ lftp
lftp :~> open -u karaf sftp://localhost:8101
Password:
lftp karaf@localhost:~> ls
-rw-r--r--   1 jbonofre jbonofre    27754 Oct 26 10:50 LICENSE
-rw-r--r--   1 jbonofre jbonofre     1919 Dec  3 05:34 NOTICE
-rw-r--r--   1 jbonofre jbonofre     3933 Aug 18  2012 README
-rw-r--r--   1 jbonofre jbonofre   101041 Dec  3 05:34
RELEASE-NOTES
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 12:51 bin
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 18:57 data
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 12:51 demos
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 13:02 deploy
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 17:59 etc
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 13:02 instances
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 13:02 lib
-rw-r--r--   1 jbonofre jbonofre        0 Dec  3 13:02 lock
drwxr-xr-x   1 jbonofre jbonofre     4096 Dec  3 12:51 system
lftp karaf@localhost:/>
```

You can also use graphic client like `filezilla`, `gftp`, `nautilus`, etc.

On Windows, you can use `filezilla`, `WinSCP`, etc.

Apache Maven

Apache Karaf `system` folder is the Karaf repository, that use a Maven directory structure. It's where Apache Karaf
looks for the artifacts (bundles, features, kars, etc).

Using Apache Maven, you can populate the `system` folder using the `deploy:deploy-file` goal.

For instance, you want to add the Apache ServiceMix facebook4j OSGi bundle, you can do:

```
mvn deploy:deploy-file
-Dfile=org.apache.servicemix.bundles.facebook4j-2.0.2_1.jar
-DgroupId=org.apache.servicemix.bundles
-DartifactId=org.apache.servicemix.bundles.facebook4j
-Dversion=2.0.2_1 -Dpackaging=jar -Durl=scp://localhost:8101/
system
```

> If you want to turn Apache Karaf as a simple Maven repository, you can use Apache Karaf Cave.

JMX MBEANSERVER

Apache Karaf provides a JMX MBeanServer.

This MBeanServer is available remotely, using any JMX client like `jconsole`.

You can find details on the Monitoring section of the user guide.

Configuration

FILES

Apache Karaf stores and loads all configuration in files located in the `etc` folder.

By default, the `etc` folder is located relatively to the `KARAF_BASE` folder. You can define another location
using the `KARAF_ETC` variable.

Each configuration is identified by a ID (the ConfigAdmin PID). The configuration files name follows the `pid.cfg`
name convention.

For instance, `etc/org.apache.karaf.shell.cfg` means that this file is the file used by the configuration with
`org.apache.karaf.shell` as PID.

A configuration file is a properties file containing key/value pairs:

```
property=value
```

In Apache Karaf, a configuration is PID with a set of properties attached.

Apache Karaf automatically loads all `*.cfg` files from the `etc` folder.

You can configure the behaviour of the configuration files using some dedicated properties in the
`etc/config.properties` configuration file:

```
...
#
# Configuration FileMonitor properties
#
felix.fileinstall.enableConfigSave = true
felix.fileinstall.dir    = ${karaf.etc}
felix.fileinstall.filter = .*\\.cfg
felix.fileinstall.poll   = 1000
felix.fileinstall.noInitialDelay = true
felix.fileinstall.log.level = 3
felix.fileinstall.log.default = jul
...
```

- `felix.fileinstall.enableConfigSave` flush back in the configuration file the changes performed directly on the

configuration service (ConfigAdmin). If `true`, any change (using `config:*` commands, MBeans, OSGi service) is
persisted back in the configuration false. Default is `true`.

- `felix.fileinstall.dir` is the directory where Apache Karaf is looking for configuration files. Default is `${karaf.etc}`
  meaning the value of the `KARAF_ETC` variable.
- `felix.fileinstall.filter` is the file name pattern used to load only some configuration files. Only files matching
  the pattern will be loaded. Default value is `.*`
  `.cfg` meaning *.cfg files.
- `felix.fileinstall.poll` is the polling interval (in milliseconds). Default value is `1000` meaning that Apache
  Karaf "re-loads" the configuration files every second.
- `felix.fileinstall.noInitialDelay` is a flag indicating if the configuration file polling starts as soon as Apache
  Karaf starts or wait for a certain time. If `true`, Apache Karaf polls the configuration files as soon as the configuration
  service starts.
- `felix.fileinstall.log.level` is the log message verbosity level of the configuration polling service. More
  this value is high, more verbose the configuration service is.
- `felix.fileinstall.log.default` is the logging framework to use, `jul` meaning Java Util Logging.

You can change the configuration at runtime by directly editiing the configuration file.

You can also do the same using the `config:*` commands or the ConfigMBean.

CONFIG:* COMMANDS

Apache Karaf provides a set of commands to manage the configuration.

config:list

`config:list` displays the list of all configurations available, or the properties in a given configuration (PID).

Without the `query` argument, the `config:list` command display all configurations, with PID, attached bundle and
properties defined in the configuration:

```
karaf@root()> config:list
----------------------------------------------------------------
Pid:
```

```
org.apache.karaf.service.acl.command.system.start-level
BundleLocation: mvn:org.apache.karaf.shell/
org.apache.karaf.shell.console/4.0.0
Properties:
   service.guard =
(&(osgi.command.scope=system)(osgi.command.function=start-level))
   * = *
   start-level = admin                              # admin can set
any start level, including < 100
   start-level[/[^0-9]*/] = viewer                  # viewer can
obtain the current start level
   execute[/.*/,/[^0-9]*/] = viewer                  # viewer can
obtain the current start level
   execute = admin                                 # admin can set any
start level, including < 100
   service.pid =
org.apache.karaf.service.acl.command.system.start-level
   start-level[/.*[0-9][0-9][0-9]+.*/] = manager # manager can
set startlevels above 100
   execute[/.*/,/.*[0-9][0-9][0-9]+.*/] = manager # manager can
set startlevels above 100
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
Properties:
   service.pid = org.apache.karaf.log
   size = 500
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
...
```

The `query` argument accepts a query using a LDAP syntax.

For instance, you can display details on one specific configuration using the following filter:

```
karaf@root()> config:list "(service.pid=org.apache.karaf.log)"
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
```

```
Properties:
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   service.pid = org.apache.karaf.log
   size = 500
```

config:edit

`config:edit` is the first command to do when you want to change a configuration. `config:edit` command put you in edition mode for a given configuration.

For instance, you can edit the `org.apache.karaf.log` configuration:

```
karaf@root()> config:edit org.apache.karaf.log
```

The `config:edit` command doesn't display anything, it just puts you in configuration edit mode. You are now ready to use other config commands (like `config:property-append`, `config:property-delete`, `config:property-set`, ...).

If you provide a configuration PID that doesn't exist yet, Apache Karaf will create a new configuration (and so a new configuration file) automatically.

All changes that you do in configuration edit mode are store in your console session: the changes are not directly applied in the configuration. It allows you to "commit" the changes (see `config:update` command) or "rollback" and cancel your changes (see `config:cancel` command).

config:property-list

The `config:property-list` lists the properties for the currently edited configuration.

Assuming that you edited the `org.apache.karaf.log` configuration, you can do:

```
karaf@root()> config:property-list
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   service.pid = org.apache.karaf.log
   size = 500
```

config:property-set

The `config:property-set` command update the value of a given property in the currently edited configuration.

For instance, to change the value of the `size` property of previously edited `org.apache.karaf.log` configuration,
you can do:

```
karaf@root()> config:property-set size 1000
karaf@root()> config:property-list
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   service.pid = org.apache.karaf.log
   size = 1000
```

If the property doesn't exist, the `config:property-set` command creates the property.

You can use `config:property-set` command outside the configuration edit mode, by specifying the `-p` (for configuration pid) option:

```
karaf@root()> config:property-set -p org.apache.karaf.log size
1000
karaf@root()> config:list "(service.pid=org.apache.karaf.log)"
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
Properties:
   service.pid = org.apache.karaf.log
   size = 1000
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
```

> Using the `pid` option, you bypass the configuration commit and rollback mechanism.

config:property-append

The `config:property-append` is similar to `config:property-set` command, but instead of completely replacing the
property value, it appends a string at the end of the property value.

For instance, to add 1 at the end of the value of the `size` property in `org.apache.karaf.log` configuration
(and so have 5001 for the value instead of 500), you can do:

```
karaf@root()> config:property-append size 1
karaf@root()> config:property-list
   service.pid = org.apache.karaf.log
   size = 5001
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
```

Like the `config:property-set` command, if the property doesn't exist, the `config:property-set` command creates
the property.

You can use the `config:property-append` command outside the configuration edit mode, by specifying the `-p` (for configuration pid) option:

```
karaf@root()> config:property-append -p org.apache.karaf.log
size 1
karaf@root()> config:list "(service.pid=org.apache.karaf.log)"
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
Properties:
   service.pid = org.apache.karaf.log
   size = 5001
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
```

Using the `pid` option, you bypass the configuration commit and rollback mechanism.

`config:property-delete`

The `config:property-delete` command delete a property in the currently edited configuration.

For instance, you previously added a `test` property in `org.apache.karaf.log` configuration. To delete this `test` property, you do:

```
karaf@root()> config:property-set test test
karaf@root()> config:property-list
   service.pid = org.apache.karaf.log
   size = 500
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
   test = test
karaf@root()> config:property-delete test
karaf@root()> config:property-list
   service.pid = org.apache.karaf.log
   size = 500
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
```

You can use the `config:property-delete` command outside the configuration edit mode, by specifying the `-p` (for configuration pid) option:

```
karaf@root()> config:property-delete -p org.apache.karaf.log test
```

`config:update` and `config:cancel`

When you are in the configuration edit mode, all changes that you do using `config:property*` commands are stored in "memory" (actually in the console session).

Thanks to that, you can "commit" your changes using the `config:update` command. The `config:update` command will commit your changes, update the configuration, and (if possible) update the configuration files.

For instance, after changing `org.apache.karaf.log` configuration with some `config:property*` commands, you have to commit your change like this:

```
karaf@root()> config:edit org.apache.karaf.log
karaf@root()> config:property-set test test
karaf@root()> config:update
karaf@root()> config:list "(service.pid=org.apache.karaf.log)"
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
Properties:
   service.pid = org.apache.karaf.log
   size = 500
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
   test = test
```

On the other hand, if you want to "rollback" your changes, you can use the `config:cancel` command. It will cancel all changes that you did, and return of the configuration state just before the `config:edit` command. The `config:cancel` exits from the edit mode.

For instance, you added the test property in the `org.apache.karaf.log` configuration, but it was a mistake:

```
karaf@root()> config:edit org.apache.karaf.log
karaf@root()> config:property-set test test
karaf@root()> config:cancel
karaf@root()> config:list "(service.pid=org.apache.karaf.log)"
----------------------------------------------------------------
Pid:            org.apache.karaf.log
BundleLocation: mvn:org.apache.karaf.log/
org.apache.karaf.log.core/4.0.0
Properties:
   service.pid = org.apache.karaf.log
   size = 500
   pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
   felix.fileinstall.filename = file:/opt/apache-karaf-4.0.0/etc/
org.apache.karaf.log.cfg
```

config:delete

The `config:delete` command completely delete an existing configuration. You don't have to be in edit mode to delete
a configuration.

For instance, you added `my.config` configuration:

```
karaf@root()> config:edit my.config
karaf@root()> config:property-set test test
karaf@root()> config:update
karaf@root()> config:list "(service.pid=my.config)"
----------------------------------------------------------------
Pid:            my.config
BundleLocation: null
Properties:
   service.pid = my.config
   test = test
```

You can delete the `my.config` configuration (including all properties in the configuration) using the `config:delete`
command:

```
karaf@root()> config:delete my.config
karaf@root()> config:list "(service.pid=my.config)"
karaf@root()>
```

config:meta

The `config:meta` command lists the meta type information related to a given configuration.

It allows you to get details about the configuration properties: key, name, type, default value, and description:

```
karaf@root()> config:meta -p org.apache.karaf.log
Meta type informations for pid: org.apache.karaf.log
key     | name    | type   |
default
| description
------------------------------------------------------------------------------
size    | Size    | int    |
500
| size of the log to keep in memory
pattern | Pattern | String | %d{ABSOLUTE} | %-5.5p | %-16.16t |
```

```
%-32.32c{1} | %-32.32C %4L | %m%n | Pattern used to display log
entries
```

JMX CONFIGMBEAN

On the JMX layer, you have a MBean dedicated to the management of the configurations: the ConfigMBean.

The ConfigMBean object name is: `org.apache.karaf:type=config,name=*`.

Attributes

The `Configs` attribute is a list of all configuration PIDs.

Operations

- `listProperties(pid)` returns the list of properties (property=value formatted) for the configuration `pid`.
- `deleteProperty(pid, property)` deletes the `property` from the configuration `pid`.
- `appendProperty(pid, property, value)` appends `value` at the end of the value of the `property` of the configuration `pid`.
- `setProperty(pid, property, value)` sets `value` for the value of the `property` of the configuration `pid`.
- `delete(pid)` deletes the configuration identified by the `pid`.
- `create(pid)` creates an empty (without any property) configuration with `pid`.
- `update(pid, properties)` updates a configuration identified with `pid` with the provided `properties` map.

Artifacts repositories and URLs

The main information provided by a feature is the set of OSGi bundles that defines the application. Such bundles are URLs pointing to the actual bundle jars. For example, one would write the following definition:

```
<bundle>http://repo1.maven.org/maven2/org/apache/servicemix/nmr/
org.apache.servicemix.nmr.api/1.0.0-m2/
org.apache.servicemix.nmr.api-1.0.0-m2.jar</bundle>
```

Doing this will make sure the above bundle is installed while installing the feature.

However, Karaf provides several URL handlers, in addition to the usual ones (file, http, etc...). One of these is the Maven URL handler, which allow reusing maven repositories to point to the bundles.

You can deploy bundles from file system without using Maven

As we can use file: as protocol handler to deploy bundles, you can use the following syntax to deploy bundles when they are
located in a directory which is not available using Maven

```
<bundle>file:base/bundles/
org.apache.servicemix.nmr.api-1.0.0-m2.jar</bundle>
```

Note: The path is relative to the Apache Karaf installation directory

Maven URL Handler

The equivalent of the above bundle would be:

```
<bundle>mvn:org.apache.servicemix.nmr/
org.apache.servicemix.nmr.api/1.0.0-m2</bundle>
```

In addition to being less verbose, the Maven url handlers can also resolve snapshots and can use a local copy of the jar if one is available in your Maven local repository.

The `org.ops4j.pax.url.mvn` bundle resolves `mvn` URLs. It can be configured using the file `etc/org.ops4j.pax.url.cfg`

The most important property is :

- `org.ops4j.pax.url.mvn.repositories` : Comma separated list of repository remote repository URLs that are checked in order of occurence when resolving maven artifacts

By default, snapshots are disabled. To enable an URL for snapshots append @snapshots to a repository entry. For example

```
http://www.example.org/repo@snapshots
```

Repositories on the local machine are supported through `file:/` URLs.

Provisioning

Apache Karaf supports the provisioning of applications and modules using the concept of Karaf Features.

APPLICATION

By provisioning application, it means install all modules, configuration, and transitive applications.

OSGI

It natively supports the deployment of OSGi applications.

An OSGi application is a set of OSGi bundles. An OSGi bundles is a regular jar file, with additional metadata in the jar MANIFEST.

In OSGi, a bundle can depend to other bundles. So, it means that to deploy an OSGi application, most of the time, you have
to firstly deploy a lot of other bundles required by the application.

So, you have to find these bundles first, install the bundles. Again, these "dependency" bundles may require other bundles
to satisfy their own dependencies.

More over, typically, an application requires configuration (see the Configuration section of the user guide).
So, before being able to start your application, in addition of the dependency bundles, you have to create or deploy the
configuration.

As we can see, the provisioning of an application can be very long and fastidious.

FEATURE AND RESOLVER

Apache Karaf provides a simple and flexible way to provision applications.

In Apache Karaf, the application provisioning is an Apache Karaf "feature".

A feature describes an application as:

- a name
- a version
- a optional description (eventually with a long description)
- a set of bundles

- optionally a set configurations or configuration files
- optionally a set of dependency features

When you install a feature, Apache Karaf installs all resources described in the feature. It means that it will
automatically resolves and installs all bundles, configurations, and dependency features described in the feature.

The feature resolver checks the service requirements, and install the bundles providing the services matching the requirements.
The default mode enables this behavior only for "new style" features repositories (basically, the features repositories XML with
schema equal or greater to 1.3.0). It doesn't apply for "old style" features repositories (coming from Karaf 2 or 3).

You can change the service requirements enforcement mode in `etc/org.apache.karaf.features.cfg` file, using the `serviceRequirements` property.

```
serviceRequirements=default
```

The possible values are:

- disable: service requirements are completely ignored, for both "old style" and "new style" features repositories
- default: service requirements are ignored for "old style" features repositories, and enabled for "new style" features repositories.
- enforce: service requirements are always verified, for "old style" and "new style" features repositories.

Additionally, a feature can also define requirements. In that case, Karaf can automatically additional bundles or features
providing the capabilities to satisfy the requirements.

A feature has a complete lifecycle: install, start, stop, update, uninstall.

FEATURES REPOSITORIES

The features are described in a features XML descriptor. This XML file contains the description of a set of features.

A features XML descriptor is named a "features repository". Before being able to install a feature, you have to register
the features repository that provides the feature (using `feature:repo-add` command or FeatureMBean as described later).

For instance, the following XML file (or "features repository") describes the `feature1` and `feature2` features:

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.3.0">
  <feature name="feature1" version="1.0.0">
    <bundle>...</bundle>
    <bundle>...</bundle>
  </feature>
  <feature name="feature2" version="1.1.0">
    <feature>feature1</feature>
    <bundle>...</bundle>
  </feature>
</features>
```

We can note that the features XML has a schema. Take a look on Features XML Schema section of the user guide
for details.
The `feature1` feature is available in version `1.0.0`, and contains two bundles. The `<bundle/>` element contains a URL
to the bundle artifact (see Artifacts repositories and URLs section for details). If you install the `feature1` feature
(using `feature:install` or the FeatureMBean as described later), Apache Karaf will automatically installs the two bundles
described.
The `feature2` feature is available in version `1.1.0`, and contains a reference to the `feature1` feature and a bundle.
The `<feature/>` element contains the name of a feature. A specific feature version can be defined using the `version`
attribute to the `<feature/>` element (`<feature version="1.0.0">feature1</feature>`). If the `version` attribute is
not specified, Apache Karaf will install the latest version available. If you install the `feature2` feature (using `feature:install`
or the FeatureMBean as described later), Apache Karaf will automatically installs `feature1` (if it's not already installed)
and the bundle.

A feature repository is registered using the URL to the features XML file.

The features state is stored in the Apache Karaf cache (in the `KARAF_DATA` folder). You can restart Apache Karaf, the
previously installed features remain installed and available after restart.
If you do a clean restart or you delete the Apache Karaf cache (delete the `KARAF_DATA` folder), all previously features
repositories registered and features installed will be lost: you will have to register the features repositories and install
features by hand again.
To prevent this behaviour, you can specify features as boot features.

BOOT FEATURES

You can describe some features as boot features. A boot feature will be automatically install by Apache Karaf, even if it has
not been previously installed using `feature:install` or FeatureMBean.

Apache Karaf features configuration is located in the `etc/org.apache.karaf.features.cfg` configuration file.

This configuration file contains the two properties to use to define boot features:

- `featuresRepositories` contains a list (coma separated) of features repositories (features XML) URLs.
- `featuresBoot` contains a list (come separated) of features to install at boot.

FEATURES UPGRADE

You can update a release by installing the same feature (with the same SNAPSHOT version or a different version).

Thanks to the features lifecycle, you can control the status of the feature (started, stopped, etc).

You can also use a simulation to see what the update will do.

FEATURE BUNDLES

Start Level

By default, the bundles deployed by a feature will have a start-level equals to the value defined in the `etc/config.properties` configuration file, in the `karaf.startlevel.bundle` property.

This value can be "overrided" by the `start-level` attribute of the `<bundle/>` element, in the features XML.

```
  <feature name="my-project" version="1.0.0">
    <bundle start-level="80">mvn:com.mycompany.myproject/
myproject-dao</bundle>
    <bundle start-level="85">mvn:com.mycompany.myproject/
myproject-service</bundle>
  </feature>
```

The start-level attribute insure that the `myproject-dao` bundle is started before the bundles that use it.

Instead of using start-level, a better solution is to simply let the OSGi framework know what your dependencies are by
defining the packages or services you need. It is more robust than setting start levels.

Simulate, Start and stop

You can simulate the installation of a feature using the `-t` option to `feature:install` command.

You can install a bundle without starting it. By default, the bundles in a feature are automatically started.

A feature can specify that a bundle should not be started automatically (the bundle stays in resolved state).
To do so, a feature can specify the `start` attribute to false in the `<bundle/>` element:

```
  <feature name="my-project" version="1.0.0">
    <bundle start-level="80"
start="false">mvn:com.mycompany.myproject/myproject-dao</bundle>
    <bundle start-level="85"
start="false">mvn:com.mycompany.myproject/
myproject-service</bundle>
  </feature>
```

Dependency

A bundle can be flagged as being a dependency, using the `dependency` attribute set to true on the `<bundle/>` element.

This information can be used by resolvers to compute the full list of bundles to be installed.

DEPENDENT FEATURES

A feature can depend to a set of other features:

```
  <feature name="my-project" version="1.0.0">
    <feature>other</feature>
    <bundle start-level="80"
start="false">mvn:com.mycompany.myproject/myproject-dao</bundle>
    <bundle start-level="85"
start="false">mvn:com.mycompany.myproject/
myproject-service</bundle>
  </feature>
```

When the `my-project` feature will be installed, the `other` feature will be automatically installed as well.

It's possible to define a version range for a dependent feature:

```
<feature name="spring-dm">
  <feature version="[2.5.6,4)">spring</feature>
  ...
</feature>
```

The feature with the highest version available in the range will be installed.

If a single version is specified, this version will be chosen.

If nothing is specified, the highest available will be installed.

FEATURE CONFIGURATIONS

The `<config/>` element in a feature XML allows a feature to create and/or populate a configuration (identified by a configuration PID).

```
<config name="com.foo.bar">
  myProperty = myValue
</config>
```

The `name` attribute of the `<config/>` element corresponds to the configuration PID (see the Configuration section for details).

The installation of the feature will have the same effect as dropping a file named `com.foo.bar.cfg` in the `etc` folder.

The content of the `<config/>` element is a set of properties, following the key=value standard.

FEATURE CONFIGURATION FILES

Instead of using the `<config/>` element, a feature can specify `<configfile/>` elements.

```
<configfile finalname="/etc/myfile.cfg"
override="false">URL</configfile>
```

Instead of directly manipulating the Apache Karaf configuration layer (as when using the `<config/>` element), the
`<configfile/>` element takes directly a file specified by a URL, and copy the file in

the location specified by the
`finalname` attribute.

If not specified, the location is relative from the `KARAF_BASE` variable. It's also possible to use variable like
${karaf.home}, ${karaf.base}, ${karaf.etc}, or even system properties.

For instance:

```
<configfile finalname="${karaf.etc}/myfile.cfg"
override="false">URL</configfile>
```

If the file is already present at the desired location it is kept and the deployment of the configuration file is skipped,
as a already existing file might contain customization. This behaviour can be overriden by `override` set to true.

The file URL is any URL supported by Apache Karaf (see the Artifacts repositories and URLs of the user guide for details).

Requirements

A feature can also specify expected requirements. The feature resolver will try to satisfy the requirements. For that, it checks
the features and bundles capabilities and will automatically install the bundles to satisfy the requirements.

For instance, a feature can contain:

```
<requirement>osgi.ee;filter:="&osgi.ee=JavaSE)(!(version>1.8)))"/requiremer
```

The requirement specifies that the feature will work by only if the JDK version is not 1.8 (so basically 1.7).

The features resolver is also able to refresh the bundles when an optional dependency is satisfy, rewiring the optional import.

COMMANDS

`feature:repo-list`

The `feature:repo-list` command lists all registered features repository:

```
karaf@root()> feature:repo-list
Repository                | URL
--------------------------------------------------------------------------------
org.ops4j.pax.cdi-0.12.0 | mvn:org.ops4j.pax.cdi/
```

```
pax-cdi-features/0.12.0/xml/features
org.ops4j.pax.web-4.1.4  | mvn:org.ops4j.pax.web/
pax-web-features/4.1.4/xml/features
standard-4.0.0           | mvn:org.apache.karaf.features/
standard/4.0.0/xml/features
enterprise-4.0.0         | mvn:org.apache.karaf.features/
enterprise/4.0.0/xml/features
spring-4.0.0             | mvn:org.apache.karaf.features/spring/
4.0.0/xml/features
```

Each repository has a name and the URL to the features XML.

Apache Karaf parses the features XML when you register the features repository URL (using `feature:repo-add` command or the FeatureMBean as described later). If you want to force Apache Karaf to reload the features repository URL (and so update the features definition), you can use the `-r` option:

```
karaf@root()> feature:repo-list -r
Reloading all repositories from their urls

Repository               | URL
-------------------------------------------------------------------------------
org.ops4j.pax.cdi-0.12.0 | mvn:org.ops4j.pax.cdi/
pax-cdi-features/0.12.0/xml/features
org.ops4j.pax.web-4.1.4  | mvn:org.ops4j.pax.web/
pax-web-features/4.1.4/xml/features
standard-4.0.0           | mvn:org.apache.karaf.features/
standard/4.0.0/xml/features
enterprise-4.0.0         | mvn:org.apache.karaf.features/
enterprise/4.0.0/xml/features
spring-4.0.0             | mvn:org.apache.karaf.features/spring/
4.0.0/xml/features
```

`feature:repo-add`

To register a features repository (and so having new features available in Apache Karaf), you have to use the `feature:repo-add` command.

The `feature:repo-add` command requires the `name/url` argument. This argument accepts:

- a feature repository URL. It's an URL directly to the features XML file. Any URL described in the Artifacts repositories and URLs section of the user guide is supported.

- a feature repository name defined in the `etc/org.apache.karaf.features.repos.cfg` configuration file.

The `etc/org.apache.karaf.features.repos.cfg` defines a list of "pre-installed/available" features repositories:

```
################################################################################
#
#    Licensed to the Apache Software Foundation (ASF) under one
or more
#    contributor license agreements.  See the NOTICE file
distributed with
#    this work for additional information regarding copyright
ownership.
#    The ASF licenses this file to You under the Apache License,
Version 2.0
#    (the "License"); you may not use this file except in
compliance with
#    the License.  You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing,
software
#    distributed under the License is distributed on an "AS IS"
BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied.
#    See the License for the specific language governing
permissions and
#    limitations under the License.
#
################################################################################

#
# This file describes the features repository URL
# It could be directly installed using feature:repo-add command
#
enterprise=mvn:org.apache.karaf.features/enterprise/LATEST/xml/
features
spring=mvn:org.apache.karaf.features/spring/LATEST/xml/features
cellar=mvn:org.apache.karaf.cellar/apache-karaf-cellar/LATEST/
xml/features
cave=mvn:org.apache.karaf.cave/apache-karaf-cave/LATEST/xml/
```

```
features
camel=mvn:org.apache.camel.karaf/apache-camel/LATEST/xml/features
camel-extras=mvn:org.apache-extras.camel-extra.karaf/camel-extra/
LATEST/xml/features
cxf=mvn:org.apache.cxf.karaf/apache-cxf/LATEST/xml/features
cxf-dosgi=mvn:org.apache.cxf.dosgi/cxf-dosgi/LATEST/xml/features
cxf-xkms=mvn:org.apache.cxf.services.xkms/
cxf-services-xkms-features/LATEST/xml
activemq=mvn:org.apache.activemq/activemq-karaf/LATEST/xml/
features
jclouds=mvn:org.apache.jclouds.karaf/jclouds-karaf/LATEST/xml/
features
openejb=mvn:org.apache.openejb/openejb-feature/LATEST/xml/
features
wicket=mvn:org.ops4j.pax.wicket/features/LATEST/xml/features
hawtio=mvn:io.hawt/hawtio-karaf/LATEST/xml/features
pax-cdi=mvn:org.ops4j.pax.cdi/pax-cdi-features/LATEST/xml/
features
pax-jdbc=mvn:org.ops4j.pax.jdbc/pax-jdbc-features/LATEST/xml/
features
pax-jpa=mvn:org.ops4j.pax.jpa/pax-jpa-features/LATEST/xml/
features
pax-web=mvn:org.ops4j.pax.web/pax-web-features/LATEST/xml/
features
pax-wicket=mvn:org.ops4j.pax.wicket/pax-wicket-features/LATEST/
xml/features
ecf=http://download.eclipse.org/rt/ecf/latest/site.p2/
karaf-features.xml
decanter=mvn:org.apache.karaf.decanter/apache-karaf-decanter/
LATEST/xml/features
```

You can directly provide a features repository name to the `feature:repo-add` command. For install, to install Apache Karaf Cellar, you can do:

```
karaf@root()> feature:repo-add cellar
Adding feature url mvn:org.apache.karaf.cellar/
apache-karaf-cellar/LATEST/xml/features
```

When you don't provide the optional `version` argument, Apache Karaf installs the latest version of the features repository available.
You can specify a target version with the `version` argument:

```
karaf@root()> feature:repo-add cellar 4.0.0.RC1
Adding feature url mvn:org.apache.karaf.cellar/
apache-karaf-cellar/4.0.0.RC1/xml/features
```

Instead of providing a features repository name defined in the `etc/org.apache.karaf.features.repos.cfg` configuration file, you can directly provide the features repository URL to the `feature:repo-add` command:

```
karaf@root()> feature:repo-add mvn:org.apache.karaf.cellar/
apache-karaf-cellar/4.0.0.RC1/xml/features
Adding feature url mvn:org.apache.karaf.cellar/
apache-karaf-cellar/4.0.0.RC1/xml/features
```

By default, the `feature:repo-add` command just registers the features repository, it doesn't install any feature.
If you specify the `-i` option, the `feature:repo-add` command registers the features repository and installs all features described in this features repository:

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

`feature:repo-refresh`

Apache Karaf parses the features repository XML when you register it (using `feature:repo-add` command or the FeatureMBean).
If the features repository XML changes, you have to indicate to Apache Karaf to refresh the features repository to load the changes.

The `feature:repo-refresh` command refreshes the features repository.

Without argument, the command refreshes all features repository:

```
karaf@root()> feature:repo-refresh
Refreshing feature url mvn:org.ops4j.pax.cdi/pax-cdi-features/
0.12.0/xml/features
Refreshing feature url mvn:org.ops4j.pax.web/pax-web-features/
4.1.4/xml/features
Refreshing feature url mvn:org.apache.karaf.features/standard/
4.0.0/xml/features
Refreshing feature url mvn:org.apache.karaf.features/enterprise/
4.0.0/xml/features
Refreshing feature url mvn:org.apache.karaf.features/spring/
4.0.0/xml/features
```

Instead of refreshing all features repositories, you can specify the features repository to refresh, by providing the URL
or the features repository name (and optionally version):

```
karaf@root()> feature:repo-refresh mvn:org.apache.karaf.features/
standard/4.0.0/xml/features
Refreshing feature url mvn:org.apache.karaf.features/standard/
4.0.0/xml/features

karaf@root()> feature:repo-refresh cellar
Refreshing feature url mvn:org.apache.karaf.cellar/
apache-karaf-cellar/LATEST/xml/features
```

feature:repo-remove

The `feature:repo-remove` command removes a features repository from the registered ones.

The `feature:repo-remove` command requires a argument:

- the features repository name (as displayed in the repository column of the `feature:repo-list` command output)
- the features repository URL (as displayed in the URL column of the `feature:repo-list` command output)

```
karaf@root()> feature:repo-remove karaf-cellar-4.0.0.RC1

karaf@root()> feature:repo-remove mvn:org.apache.karaf.cellar/
apache-karaf-cellar/LATEST/xml/features
```

By default, the `feature:repo-remove` command just removes the features repository from the registered ones: it doesn't
uninstall the features provided by the features repository.

If you use `-u` option, the `feature:repo-remove` command uninstalls all features described by the features repository:

```
karaf@root()> feature:repo-remove -u karaf-cellar-4.0.0.RC1
```

feature:list

The `feature:list` command lists all available features (provided by the different registered features repositories):

```
Name                          | Version
| Required | State       | Repository               | Description
```

```
---------------------------------------------------------------------------
pax-cdi                          | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI support
pax-cdi-1.1                      | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI 1.1 support
pax-cdi-1.2                      | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI 1.2 support
pax-cdi-weld                     | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI
support
pax-cdi-1.1-weld                 | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI
1.1 support
pax-cdi-1.2-weld                 | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI
1.2 support
pax-cdi-openwebbeans             | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 |
OpenWebBeans CDI support
pax-cdi-web                      | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI
support
pax-cdi-1.1-web                  | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI
1.1 support
...
```

If you want to order the features by alphabetical name, you can use the `-o` option:

```
karaf@root()> feature:list -o
Name                             | Version
| Required | State       | Repository               | Description
---------------------------------------------------------------------------
deltaspike-core                  | 1.2.1
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache
Deltaspike core support
deltaspike-data                  | 1.2.1
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache
Deltaspike data support
deltaspike-jpa                   | 1.2.1
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache
```

```
Deltaspike jpa support
deltaspike-partial-bean       | 1.2.1
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache
Deltaspike partial bean support
pax-cdi                       | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI support
pax-cdi-1.1                   | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI 1.1 support
pax-cdi-1.1-web               | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI
1.1 support
pax-cdi-1.1-web-weld          | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web
CDI 1.1 support
pax-cdi-1.1-weld              | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI
1.1 support
pax-cdi-1.2                   | 0.12.0
|          | Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide
CDI 1.2 support
...
```

By default, the `feature:list` command displays all features, whatever their current state (installed or not installed).

Using the `-i` option displays only installed features:

```
karaf@root()> feature:list -i
Name            | Version | Required | State   | Repository
| Description
--------------------------------------------------------------------------------
aries-proxy     | 4.0.0   |          | Started | standard-4.0.0
| Aries Proxy
aries-blueprint | 4.0.0   | x        | Started | standard-4.0.0
| Aries Blueprint
feature         | 4.0.0   | x        | Started | standard-4.0.0
| Features Support
shell           | 4.0.0   | x        | Started | standard-4.0.0
| Karaf Shell
shell-compat    | 4.0.0   | x        | Started | standard-4.0.0
| Karaf Shell Compatibility
deployer        | 4.0.0   | x        | Started | standard-4.0.0
```

```
| Karaf Deployer
bundle           | 4.0.0   | x        | Started | standard-4.0.0
| Provide Bundle support
config           | 4.0.0   | x        | Started | standard-4.0.0
| Provide OSGi ConfigAdmin support
diagnostic       | 4.0.0   | x        | Started | standard-4.0.0
| Provide Diagnostic support
instance         | 4.0.0   | x        | Started | standard-4.0.0
| Provide Instance support
jaas             | 4.0.0   | x        | Started | standard-4.0.0
| Provide JAAS support
log              | 4.0.0   | x        | Started | standard-4.0.0
| Provide Log support
package          | 4.0.0   | x        | Started | standard-4.0.0
| Package commands and mbeans
service          | 4.0.0   | x        | Started | standard-4.0.0
| Provide Service support
system           | 4.0.0   | x        | Started | standard-4.0.0
| Provide System support
kar              | 4.0.0   | x        | Started | standard-4.0.0
| Provide KAR (KARaf archive) support
ssh              | 4.0.0   | x        | Started | standard-4.0.0
| Provide a SSHd server on Karaf
management       | 4.0.0   | x        | Started | standard-4.0.0
| Provide a JMX MBeanServer and a set of MBeans in
wrap             | 0.0.0   | x        | Started | standard-4.0.0
| Wrap URL handler
```

`feature:install`

The `feature:install` command installs a feature.

It requires the `feature` argument. The `feature` argument is the name of the feature, or the name/version of the feature.
If only the name of the feature is provided (not the version), the latest version available will be installed.

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

We can simulate an installation using `-t` or `--simulate` option: it just displays what it would do, but it doesn't do it:

```
karaf@root()> feature:install -t -v eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
```

```
No deployment change.
  Managing bundle:
    org.apache.felix.metatype / 1.0.12
```

You can specify a feature version to install:

```
karaf@root()> feature:install eventadmin/4.0.0
```

By default, the `feature:install` command is not verbose. If you want to have some details about actions performed by the `feature:install` command, you can use the `-v` option:

```
karaf@root()> feature:install -v eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
Done.
```

If a feature contains a bundle which is already installed, by default, Apache Karaf will refresh this bundle.
Sometime, this refresh can cause issue to other running applications. If you want to disable the auto-refresh of installed bundles, you can use the `-r` option:

```
karaf@root()> feature:install -v -r eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
Done.
```

You can decide to not start the bundles installed by a feature using the `-s` or `--no-auto-start` option:

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

`feature:start`

By default, when you install a feature, it's automatically installed. However, you can specify the `-s` option to the `feature:install` command.

As soon as you install a feature (started or not), all packages provided by the bundles defined in the feature will be available, and can be used for the wiring in other bundles.

When starting a feature, all bundles are started, and so, the feature also exposes the services.

feature:stop

You can also stop a feature: it means that all services provided by the feature will be stop and removed from the service registry. However, the packages
are still available for the wiring (the bundles are in resolved state).

feature:uninstall

The `feature:uninstall` command uninstalls a feature. As the `feature:install` command, the `feature:uninstall` command
requires the `feature` argument. The `feature` argument is the name of the feature, or the name/version of the feature.
If only the name of the feature is provided (not the version), the latest version available will be installed.

```
karaf@root()> feature:uninstall eventadmin
```

The features resolver is involved during feature uninstallation: transitive features installed by the uninstalled feature can be uninstalled
themselves if not used by other feature.

DEPLOYER

You can "hot deploy" a features XML by dropping the file directly in the `deploy` folder.

Apache Karaf provides a features deployer.

When you drop a features XML in the deploy folder, the features deployer does:

- register the features XML as a features repository
- the features with `install` attribute set to "auto" will be automatically installed by the features deployer.

For instance, dropping the following XML in the deploy folder will automatically install feature1 and feature2, whereas
feature3 won't be installed:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="my-features" xmlns="http://karaf.apache.org/
xmlns/features/v1.3.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
        xsi:schemaLocation="http://karaf.apache.org/xmlns/
features/v1.3.0 http://karaf.apache.org/xmlns/features/v1.3.0">

    <feature name="feature1" version="1.0" install="auto">
        ...
    </feature>
```

```
    <feature name="feature2" version="1.0" install="auto">
        ...
    </feature>

    <feature name="feature3" version="1.0">
        ...
    </feature>

</features>
```

JMX FEATUREMBEAN

On the JMX layer, you have a MBean dedicated to the management of the features and features repositories: the FeatureMBean.

The FeatureMBean object name is: `org.apache.karaf:type=feature,name=*`.

Attributes

The FeatureMBean provides two attributes:

- `Features` is a tabular data set of all features available.
- `Repositories` is a tabular data set of all registered features repositories.

The `Repositories` attribute provides the following information:

- `Name` is the name of the features repository.
- `Uri` is the URI to the features XML for this repository.
- `Features` is a tabular data set of all features (name and version) provided by this features repository.
- `Repositories` is a tabular data set of features repositories "imported" in this features repository.

The `Features` attribute provides the following information:

- `Name` is the name of the feature.
- `Version` is the version of the feature.
- `Installed` is a boolean. If true, it means that the feature is currently installed.
- `Bundles` is a tabular data set of all bundles (bundles URL) described in the feature.
- `Configurations` is a tabular data set of all configurations described in the feature.
- `Configuration Files` is a tabular data set of all configuration files described in the feature.
- `Dependencies` is a tabular data set of all dependent features described in the feature.

Operations

- `addRepository(url)` adds the features repository with the `url`. The `url` can be a `name` as in the `feature:repo-add` command.
- `addRepository(url, install)` adds the features repository with the `url` and automatically installs all bundles if `install` is true. The `url` can be a `name` like in the `feature:repo-add` command.
- `removeRepository(url)` removes the features repository with the `url`. The `url` can be a `name` as in the `feature:repo-remove` command.
- `installFeature(name)` installs the feature with the `name`.
- `installFeature(name, version)` installs the feature with the `name` and `version`.
- `installFeature(name, noClean, noRefresh)` installs the feature with the `name` without cleaning the bundles in case of failure, and without refreshing already installed bundles.
- `installFeature(name, version, noClean, noRefresh) }} installs the feature with the {{name` and `version` without cleaning the bundles in case of failure, and without refreshing already installed bundles.
- `uninstallFeature(name)` uninstalls the feature with the `name`.
- `uninstallFeature(name, version)` uninstalls the feature with the `name` and `version`.

Notifications

The FeatureMBean sends two kind of notifications (on which you can subscribe and react):

- When a feature repository changes (added or removed).
- When a feature changes (installed or uninstalled).

Developers Guide

Developer commands

As you can see in the users guide, Apache Karaf is an enterprise ready OSGi container.

It's also a container designed to simplify the life for developers and administrators to get details about the
running container.

DUMP

If you encounter issues like performance degradations, weird behaviour, it could be helpful to have a kind of snapshot
about the current activity of the container.

The `dev:dump-create` command creates a dump file containing:

- the `bundles.txt` file contains the list of all OSGi bundles, with id, symbolic name, version, current status
- the `features.txt` file contains the list of all features, including current status
- the `environment.txt` file contains details about Apache Karaf, OSGi framework, Operating System, JVM, system
  properties, threads count, classes loaded
- the `memory.txt` file contains the status of the JVM memory at the dump time
- the `heapdump.txt` file contains a memory heap dump, with all objects instances, space usage, etc.
- the `threads.txt` file contains a thread dump, with all threads, waiting status, etc.
- the `log` folder contains the `data/log` folder, with all log files.

By default, the `dev:dump-create` command creates a zip file in the `KARAF_BASE` folder, with the timestamp of the
dump creation:

```
karaf@root()> dev:dump-create
Created dump zip: 2015-07-01_171434.zip
```

We can see the file generated in the `KARAF_BASE` folder:

```
$ cd /opt/apache-karaf-4.0.0
$ ls -lh *.zip
-rw-rw-r-- 1 user group 17M Jul  1 17:14 2015-07-01_171434.zip
```

You can specify the file name of the zip archive:

```
karaf@root()> dev:dump-create mydump.zip
Diagnostic dump created.
```

Instead of a zip archive, you can create the dump (exploded) in a directory using the `-d` (`--directory`) option:

```
karaf@root()> dev:dump-create -d /tmp/mydump
Diagnostic dump created.
```

DIAGNOSTIC

It's not always easy for the developers to understand why a bundle is not active.

It could be because the Activator failed, the Blueprint container start failed, etc.

The `bundle:diag` command gives you details about a bundle is not active:

```
karaf@root()> bundle:diag
Apache ServiceMix :: Bundles :: avro-ipc (81)
---------------------------------------------
Status: Installed
Unsatisfied Requirements:
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=javax.servlet)(version>=2.5.0)(!(version>=3.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=javax.servlet.http)(version>=2.5.0)(!(version>=3.0.0
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.data)(version>=1.7.0)(!(version>=2.0
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.file)(version>=1.7.0)(!(version>=2.0
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.generic)(version>=1.7.0)(!(version>=
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.io)(version>=1.7.0)(!(version>=2.0.0
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.reflect)(version>=1.7.0)(!(version>=
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.specific)(version>=1.7.0)(!(version>
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.util)(version>=1.7.0)(!(version>=2.0
[81.0] osgi.wiring.package;
(osgi.wiring.package=org.apache.velocity)
[81.0] osgi.wiring.package;
```

```
(osgi.wiring.package=org.apache.velocity.app)
[81.0] osgi.wiring.package;
(osgi.wiring.package=org.apache.velocity.context)
[81.0] osgi.wiring.package;
(osgi.wiring.package=org.apache.velocity.exception)
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.bootstrap)(version>=3.4.0)(!(version
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.buffer)(version>=3.4.0)(!(version>=4
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel)(version>=3.4.0)(!(version>=
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel.group)(version>=3.4.0)(!(ver
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel.socket.nio)(version>=3.4.0)
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.codec.frame)(version>=3.4.0)
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.codec.oneone)(version>=3.4.0
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.execution)(version>=3.4.0)(!
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty)(version>=6.1.0)(!(version>=7.0.0)
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.bio)(version>=6.1.0)(!(version>=7.
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.nio)(version>=6.1.0)(!(version>=7.
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.servlet)(version>=6.1.0)(!(version
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.resource)(version>=6.1.0)(!(version>=7.0
```

DYNAMIC IMPORT

The `bundle:dynamic-import` command allows you to enable or disable the dynamic import of a given bundle:

```
karaf@root()> bundle:dynamic-import 50
Enabling dynamic imports on bundle org.ops4j.pax.url.wrap [50]
```

The purpose of dynamic import is to allow a bundle to be wired up to packages that may not be knwon about in advance.
When a class is requested, if it cannot be solved via the bundle's existing imports, the

dynamic import allows other
bundles to be considered for a wiring import to be added.

The `bundle:dynamic-import` command allows or doesn't allow this behaviour.

OSGI FRAMEWORK

The `system:framework` command allows to display the current OSGi framework in use, and enable/disable debugging inside the OSGi framework.

```
karaf@root()> system:framework
Current OSGi framework is felix
karaf@root()> system:framework -debug
Enabling debug for OSGi framework (felix)
karaf@root()> system:framework -nodebug
Disabling debug for OSGi framework (felix)
```

STACK TRACES PRINTOUT

The `shell:stack-traces-print` command prints the full stack trace when the execution of a command
throws an exception.

You can enable or disable this behaviour by passing true (to enable) or false (to disable) on the command on the fly:

```
karaf@root()> stack-traces-print
Printing of stacktraces set to true
karaf@root()> bundle:start
java.lang.RuntimeException: Access to system bundle 0 denied.
You can override with -f
        at
org.apache.karaf.bundle.command.BundlesCommand.assertNoSystemBundles(Bundle
        at
org.apache.karaf.bundle.command.BundlesCommand.doExecute(BundlesCommand.jav
        at
org.apache.karaf.bundle.command.BundlesCommandWithConfirmation.doExecute(Bu
        at
org.apache.karaf.shell.console.AbstractAction.execute(AbstractAction.java:3
        at
org.apache.karaf.shell.console.OsgiCommandSupport.execute(OsgiCommandSuppor
        at
org.apache.karaf.shell.commands.basic.AbstractCommand.execute(AbstractComma
        at sun.reflect.GeneratedMethodAccessor30.invoke(Unknown
```

```
Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
        at java.lang.reflect.Method.invoke(Method.java:601)
        at
org.apache.aries.proxy.impl.ProxyHandler$1.invoke(ProxyHandler.java:54)
        at
org.apache.aries.proxy.impl.ProxyHandler.invoke(ProxyHandler.java:119)
        at
org.apache.karaf.shell.console.commands.$BlueprintCommand14083304.execute(U
Source)
        at sun.reflect.GeneratedMethodAccessor30.invoke(Unknown
Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
        at java.lang.reflect.Method.invoke(Method.java:601)
        at
org.apache.aries.proxy.impl.ProxyHandler$1.invoke(ProxyHandler.java:54)
        at
org.apache.aries.proxy.impl.ProxyHandler.invoke(ProxyHandler.java:119)
        at
org.apache.karaf.shell.console.commands.$BlueprintCommand14083304.execute(U
Source)
        at
org.apache.felix.gogo.runtime.CommandProxy.execute(CommandProxy.java:78)
        at
org.apache.felix.gogo.runtime.Closure.executeCmd(Closure.java:477)
        at
org.apache.felix.gogo.runtime.Closure.executeStatement(Closure.java:403)
        at org.apache.felix.gogo.runtime.Pipe.run(Pipe.java:108)
        at
org.apache.felix.gogo.runtime.Closure.execute(Closure.java:183)
        at
org.apache.felix.gogo.runtime.Closure.execute(Closure.java:120)
        at
org.apache.felix.gogo.runtime.CommandSessionImpl.execute(CommandSessionImpl
        at
org.apache.karaf.shell.console.impl.jline.ConsoleImpl$DelegateSession.execu
        at
org.apache.karaf.shell.console.impl.jline.ConsoleImpl.run(ConsoleImpl.java:
        at java.lang.Thread.run(Thread.java:722)
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3.doRun(Con
```

```
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3$1.run(Con
        at java.security.AccessController.doPrivileged(Native
Method)
        at
org.apache.karaf.jaas.modules.JaasHelper.doAs(JaasHelper.java:47)
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3.run(Conso
karaf@root()> stack-traces-print false
Printing of stacktraces set to false
karaf@root()> bundle:start
Error executing command: Access to system bundle 0 denied. You
can override with -f
```

BUNDLE TREE

The `bundle:tree-show` command shows the bundle dependency tree based on the wiring information of a given single bundle ID.

```
karaf@root()> bundle:tree-show 40
Bundle org.ops4j.pax.url.wrap [40] is currently ACTIVE

org.ops4j.pax.url.wrap [40]
+- org.ops4j.base.util.property [14]
+- org.ops4j.pax.url.commons [49]
|  +- org.ops4j.base.util.property [14]
|  +- org.ops4j.pax.logging.pax-logging-api [23]
|  +- org.ops4j.pax.swissbox.property [31]
|  |  +- org.ops4j.base.util.property [14]
|  |  +- org.ops4j.base.lang [41]
|  +- org.apache.felix.configadmin [43]
|  |  +- org.ops4j.pax.logging.pax-logging-api [23]
|  +- org.ops4j.base.lang [41]
+- org.ops4j.pax.logging.pax-logging-api [23]
+- org.ops4j.pax.swissbox.bnd [25]
|  +- biz.aQute.bndlib [30]
|  |  +- org.apache.servicemix.bundles.junit [36]
|  +- org.ops4j.pax.logging.pax-logging-api [23]
|  +- org.ops4j.base.lang [41]
+- org.apache.felix.configadmin [43]
+- org.ops4j.base.net [29]
|  +- org.ops4j.base.monitors [37]
```

```
|  +- org.ops4j.base.lang [41]
+- org.ops4j.base.lang [41]
```

WATCH

The `bundle:watch` command enables watching the local Maven repository for updates on bundles.
If the bundle file changes on the Maven repository, Apache Karaf will automatically update the bundle.

The `bundle:watch` allows you to configure a set of URLs to monitore. All bundles bundles whose location matches the
given URL will be automatically updated. It avoids needing to manually update the bundles or even copy the bundle to the
system folder.

> Only Maven based URLs and Maven SNAPSHOTs will actually be updated automatically.

The following command:

```
karaf@root()> bundle:watch *
```

will monitor all bundles that have a location matching mvn:* and '-SNAPSHOT' in their URL.

Scripting

In the console section of the users guide, we introduced the scripting support.

ASSIGNATION

You already know the first usage of scripting: execution of command.

```
karaf@root()> echo hello world
hello world
```

You can also assign value to session variables:

```
karaf@root()> msg = "hello world"
hello world
```

Once you have assigned a value to a variable, you can display this value using the "resolved" variable name:

```
karaf@root()> echo $msg
hello world
```

The () are execution quotes (like the backquotes when you use bash on Unix).

```
karaf@root()> ($.context bundle 1) location
mvn:org.apache.karaf.jaas/org.apache.karaf.jaas.modules/
3.0.1-SNAPSHOT
```

The `$.context` access the context variables in the current session.
We access to the `bundle` variable (an array containing all bundles), and we want to display the bundle location for
the bundle at the index 1 in the bundle array.

EXPRESSIONS

The shell has a built-in expression parser. Expressions must be enclosed with the `%(...)` syntax.

Examples:

```
karaf@root()> %(1+2)
3
karaf@root()> a = 0
0
karaf@root()> %(a+=1)
1
karaf@root()> %(a+=1)
2
karaf@root()> b=1
1
karaf@root()> %(SQRT(a^2 + b^2))
1.7320508
```

Mathematical Operators

Operator	Description
+	Additive operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator (Modulo)
^	Power operator

Boolean Operators

Operator	Description
=	Equals
==	Equals
!=	Not equals
<>	Not equals
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&&	Boolean and

\ \

Supported Functions

Function	Description
NOT(*expression*)	Boolean negation, 1 (means true) if the expression is not zero
IF(*condition*,*value_if_true*,*value_if_false*)	Returns one value if the condition evaluates to true or the other if it evaluates to false
RANDOM()	Produces a random number between 0 and 1
MIN(*e1*,*e2*)	Returns the smaller of both expressions
MAX(*e1*,*e2*)	Returns the bigger of both expressions
ABS(*expression*)	Returns the absolute (non-negative) value of the expression
ROUND(*expression*,precision)	Rounds a value to a certain number of digits, uses the current rounding mode
FLOOR(*expression*)	Rounds the value down to the nearest integer
CEILING(*expression*)	Rounds the value up to the nearest integer
LOG(*expression*)	Returns the natural logarithm (base e) of an expression
SQRT(*expression*)	Returns the square root of an expression
SIN(*expression*)	Returns the trigonometric sine of an angle (in degrees)
COS(*expression*)	Returns the trigonometric cosine of an angle (in degrees)
TAN(*expression*)	Returns the trigonometric tangens of an angle (in degrees)
SINH(*expression*)	Returns the hyperbolic sine of a value
COSH(*expression*)	Returns the hyperbolic cosine of a value
TANH(*expression*)	Returns the hyperbolic tangens of a value

RAD(*expression*)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians
DEG(*expression*)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees

Functions names are case insensitive.

Supported Constants

Constant	Description
PI	The value of *PI*, exact to 100 digits
TRUE	The value one
FALSE	The value zero

LIST, MAPS, PIPES AND CLOSURES

Using [], you can define array variable:

```
karaf@root()> list = [1 2 a b]
1
2
a
b
```

You can also create a map if you put variables assignation in the array:

```
karaf@root()> map = [Jan=1 Feb=2 Mar=3]
Jan                 1
Feb                 2
Mar                 3
```

Using the | character, you can pipe output from a command as an input to another one.

For instance, you can access to the bundles context variables and send it as input to the grep command:

```
karaf@root()> ($.context bundles) | grep -i felix
    0|Active     |    0|org.apache.felix.framework (4.2.1)
   21|Active     |   11|org.apache.felix.fileinstall (3.2.6)
```

```
   43|Active     |   10|org.apache.felix.configadmin (1.6.0)
   51|Active     |   30|org.apache.felix.gogo.runtime (0.10.0)
```

You can assign name to script execution. It's what we use for alias:

```
karaf@root()> echo2 = { echo xxx $args yyy }
echo xxx $args yyy
karaf@root()> echo2 hello world
xxx hello world yyy
```

STARTUP

The `etc/shell.init.script` file is executed at startup in each shell session, allowing the definition of additional
variables or aliases or even complex functions. It's like the bashrc or profile on Unix.

CONSTANTS AND VARIABLES

Apache Karaf console provides a set of implicit constants and variables that you can use in your script.

- `$.context` to access a bundle context
- `$.variables` to access the list of defined variables
- `$.commands` to access the list of defined commands

The variables starting with a # that are defined as Function (such as closures) will be executed automatically:

```
karaf@root> \#inc = { var = "${var}i" ; $var }
var = "${var}i" ; $var
karaf@root> echo $inc
i
karaf@root> echo $inc
ii
karaf@root>
```

BUILT-IN VARIABLES AND COMMANDS

Apache Karaf console provides built-in variable very useful for scripting:

- `$args` retrieves the list of script parameters, given to the closure being executed
- `$1 .. $999` retrieves the nth argument of the closure
- `$it` (same as `$1`) is used in a loop to access the current iterator value

Apache Karaf console provides commands for scripting:

- `shell:if`
- `shell:new`
- `shell:each`
- ...

See the full list of `shell` commands.

LEVERAGING EXISTING JAVA CAPABILITIES (VIA REFLECTION)

Apache Karaf console supports loading and execution of Java classes.

The `$karaf.lastException` implicit variable contains the latest Exception thrown.

```
karaf@root()> ($.context bundle) loadClass foo
Error executing command: foo not found by
org.apache.karaf.shell.console [17]
karaf@root()> $karaf.lastException printStackTrace
java.lang.ClassNotFoundException: foo not found by
org.apache.karaf.shell.console [17]
        at
org.apache.felix.framework.BundleWiringImpl.findClassOrResourceByDelegation
        at
org.apache.felix.framework.BundleWiringImpl.access$400(BundleWiringImpl.jav
        at
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.loadClass(Bun
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at
org.apache.felix.framework.Felix.loadBundleClass(Felix.java:1723)
        at
org.apache.felix.framework.BundleImpl.loadClass(BundleImpl.java:926)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:3
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
        at java.lang.reflect.Method.invoke(Method.java:597)
        at
org.apache.felix.gogo.runtime.Reflective.invoke(Reflective.java:137)
        at
org.apache.felix.gogo.runtime.Closure.executeMethod(Closure.java:527)
        at
org.apache.felix.gogo.runtime.Closure.executeStatement(Closure.java:403)
        at org.apache.felix.gogo.runtime.Pipe.run(Pipe.java:108)
```

```
        at
org.apache.felix.gogo.runtime.Closure.execute(Closure.java:183)
        at
org.apache.felix.gogo.runtime.Closure.execute(Closure.java:120)
        at
org.apache.felix.gogo.runtime.CommandSessionImpl.execute(CommandSessionImpl
        at
org.apache.karaf.shell.console.jline.Console.run(Console.java:166)
        at java.lang.Thread.run(Thread.java:680)
```

It's possible to create objects to create commands "on the fly":

```
karaf@root()> addcommand system (($.context bundle) loadClass
java.lang.System)
karaf@root()> system:getproperty karaf.name
root
```

It means that you can create object using the `new` directive, and call methods on the objects:

```
karaf@root> map = (new java.util.HashMap)
karaf@root> $map put 0 0
karaf@root> $map
0                   0
```

EXAMPLES

The following examples show some scripts defined in `etc/shell.init.script`.

The first example show a script to add a value into a configuration list:

```
#
# Add a value at the end of a property in the given OSGi
configuration
#
# For example:
# > config-add-to-list org.ops4j.pax.url.mvn
org.ops4j.pax.url.mvn.repositories http://scala-tools.org/
repo-releases
#
config-add-to-list = {
  config:edit $1 ;
  a = (config:property-list | grep --color never $2 | tac) ;
  b = (echo $a | grep --color never "\b$3\b" | tac) ;
```

```
  if { ($b trim) isEmpty } {
    if { $a isEmpty } {
      config:property-set $2 $3
    } {
      config:property-append $2 ", $3"
    } ;
    config:update
  } {
    config:cancel
  }
}
```

This second example shows a script to wait for an OSGi service, up to a given timeout, and combine this script in
other scripts:

```
#
# Wait for the given OSGi service to be available
#
wait-for-service-timeout = {
  _filter = $.context createFilter $1 ;
  _tracker = shell:new org.osgi.util.tracker.ServiceTracker
$.context $_filter null ;
  $_tracker open ;
  _service = $_tracker waitForService $2 ;
  $_tracker close
}
#
# Wait for the given OSGi service to be available with a timeout
of 10 seconds
#
wait-for-service = {
  wait-for-service-timeout $1 10000
}
#
# Wait for the given command to be available with a timeout of
10 seconds
# For example:
# > wait-for-command dev watch
#
wait-for-command = {
  wait-for-service
"(&(objectClass=org.apache.felix.service.command.Function)(osgi.command.sco
}
```

Programmatically connect

As described in the users guide, Apache Karaf supports remote access to both the console (by embedding a SSHd server) and
the management layer.

TO THE CONSOLE

You can write a Apache Karaf remote console client in Java (or other language).

Accessing to a remote Apache Karaf console means writing a SSH client. This SSH client can be in pure Java or in
another language.

For instance, the `bin/client` script starts a SSH client written in Java.

The following code is a simple code to create a SSH client:

```
import org.apache.sshd.ClientChannel;
import org.apache.sshd.ClientSession;
import org.apache.sshd.SshClient;
import org.apache.sshd.client.future.ConnectFuture;

public class Main {

    public static void main(String[] args) throws Exception {
        String host = "localhost";
        int port = 8101;
        String user = "karaf";
        String password = "karaf";

        SshClient client = null;
        try {
            client = SshClient.setUpDefaultClient();
            client.start();
            ConnectFuture future = client.connect(host, port);
            future.await();
            ClientSession session = future.getSession();
            session.authPassword(user, password);
            ClientChannel channel =
session.createChannel("shell");
            channel.setIn(System.in);
```

```
            channel.setOut(System.out);
            channel.setErr(System.err);
            channel.open();
            channel.waitFor(ClientChannel.CLOSED, 0);
        } catch (Throwable t) {
            t.printStackTrace();
            System.exit(1);
        } finally {
            try {
                client.stop();
            } catch (Throwable t) { }
        }
        System.exit(0);
    }

}
```

TO THE MANAGEMENT LAYER

The Apache Karaf management layer uses JMX. Apache Karaf embeds a JMX MBeanServer that you can use remotely.

In order to use the MBeanServer remotely, you have to write a JMX client.

The following example shows a simple JMX client stopping Apache Karaf remotely via the JMX layer:

```
javax.management.*;

public class Main {

    public static void main(String[] args) throws Exception {
        JMXServiceURL url = new
JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:1099/
karaf-root");
        JMXConnector connector =
JMXConnectorFactory.connect(url, null);
        MBeanServerConnection mbeanServer =
connector.getMBeanServerConnection();
        ObjectName systemMBean = new
ObjectName("org.apache.karaf:type=system,name=karaf-root");
        mbeanServer.invoke(systemMBean, "halt", null, null);
        connector.close();
    }
```

```
}
```

Branding

CONSOLE

You can "brand" the Apache Karaf console.

By branding, it means that you can define your own:

- the welcome message (motd or Message Of The Day) displayed when you start the console
- the prompt displayed to the users

There are 2 ways of branding the Karaf console: (1) adding a branding.properties file to etc, and (2) creating a branding bundle.

ADDING A BRANDING.PROPERTIES FILE TO ETC

Create a `etc/branding.properties` file similar to:

```
{

}
welcome = \
\u001B[36m                __ __                               ____      \u001B[0m\r\n\
\u001B[36m               / //_/____ _____________ _/ __/      \u001B[0m\r\n\
\u001B[36m              / ,<  / __ `/ ___/ __ `/ /_        \u001B[0m\r\n\
\u001B[36m             / /| |/ /_/ / /  / /_/ / __/        \u001B[0m\r\n\
\u001B[36m            /_/ |_|\\__,_/_/   \\__,_/_/          \u001B[0m\r\n\
\r\n\
\u001B[1m  Apache Karaf\u001B[0m (4.0.0)\r\n\
\r\n\
Hit '\u001B[1m<tab>\u001B[0m' for a list of available commands\r\
n\
   and '\u001B[1m[cmd] --help\u001B[0m' for help on a specific
command.\r\n\
Hit '\u001B[1m<ctrl-d>\u001B[0m' or '\u001B[1mosgi:shutdown\
u001B[0m' to shutdown Karaf.\r\n

prompt = \u001B[1m${USER}@${APPLICATION}\u001B[0m>
{

}
```

Start Karaf and you will see your branded Karaf console.

BRANDING BUNDLE

At startup, Apache Karaf is looking for a bundle which exports the `org.apache.karaf.branding` package, containing a `branding.properties` file.

Basically, a branding bundle is a very simple bundle, just containing a `org/apache/karaf/branding/branding.properties` file.

It's easy to create such branding bundle using Apache Maven.

The following `pom.xml` creates a branding bundle:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>your.group.id</groupId>
    <artifactId>your.branding.artifact.id</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>Your Branding Bundle Name</name>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>

<Bundle-SymbolicName>manual</bundle-SymbolicName>
                        <Import-Package>*</Import-Package>
                        <Private-Package>!*</Private-Package>
                        <Export-Package>
                            org.apache.karaf.branding
                        </Export-Package>

<Spring-Context>*;public-context:=false</Spring-Context>
```

```
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

You can put your `branding.properties` file in the project resources (`src/main/resources/org/apache/karaf/branding/branding.properties`):

```
welcome = This is my Karaf brand\r\n
prompt = Hey ${USER}>
```

For instance, the default Apache Karaf branding properties file contains:

```
welcome = \
\u001B[36m         __ __                  ____      \u001B[0m\r\n\
\u001B[36m        / //_/____ _____ _____ / __/      \u001B[0m\r\n\
\u001B[36m       / ,<  / __ `/ ___/ __ `/ /_        \u001B[0m\r\n\
\u001B[36m      / /| |/ /_/ / /  / /_/ / __/        \u001B[0m\r\n\
\u001B[36m     /_/ |_|\\__,_/_/   \\__,_/_/          \u001B[0m\r\n\
\r\n\
\u001B[1m  Apache Karaf\u001B[0m (4.0.3)\r\n\
\r\n\
Hit '\u001B[1m<tab>\u001B[0m' for a list of available commands\r\
n\
   and '\u001B[1m[cmd] --help\u001B[0m' for help on a specific
command.\r\n\
Hit '\u001B[1m<ctrl-d>\u001B[0m' or type
'\u001B[1msystem:shutdown\u001B[0m' or '\u001B[1mlogout\
u001B[0m' to shutdown Karaf.\r\n
```

As you can see, the `branding.properties` contains two properties:

- welcome is the welcome message displayed when you start Apache Karaf console.
- prompt is the string used to display the console prompt. This string supports variables:
    - $} defines the user name of the prompt. Caveat – the user name is presently static and hardcoded to "karaf",
      however you can override here with your own static user name.
    - ${{APPLICATION}} defines the Karaf instance name.

As you can see, both strings support ASCII escaped format. For instance \u001B[1m switches the foreground in bold
and \u001B[0m switch back to normal.

Some examples of customized prompt examples follow:

```
# Define a user with fancy colors
prompt = \u001B[36mmy-karaf-user\u001B[0m\u001B[1m@\u001B[0m\
u001B[34m${APPLICATION}\u001B[0m>

# Static sober prompt
prompt = my-user@my-karaf>
```

INSTALLING THE BRANDING BUNDLE

Thanks to the `pom.xml`, we can use `mvn` to build the branding bundle:

```
mvn install
```

You just have to drop the file in the `lib` directory:

```
cp branding.jar /opt/apache-karaf-4.0.0/lib/karaf-branding.jar
```

You can now start Apache Karaf to see your branded console.

WEBCONSOLE

It's also possible to brand the Apache Karaf WebConsole.

You have to create a bundle, fragment of the Apache Karaf WebConsole.

This WebConsole branding bundle contains a `META-INF/webconsole.properties` containing branding properties:

```
#
# This file contains branding properties to overwrite the default
# branding of the Apache Felix Web Console when deployed in an
# Apache Karaf application.


webconsole.brand.name = My Web Console

webconsole.product.name = My Karaf
webconsole.product.url = http://karaf.apache.org/
webconsole.product.image = /res/karaf/imgs/logo.png
```

```
webconsole.vendor.name = The Apache Software Foundation
webconsole.vendor.url = http://www.apache.org
webconsole.vendor.image = /res/karaf/imgs/logo.png

webconsole.favicon = /res/karaf/imgs/favicon.ico
webconsole.stylesheet = /res/karaf/ui/webconsole.css
```

The bundle also provides the css stylesheet and images defined in this properties file.

As for console, you can use the following `pom.xml` to create the WebConsole branding bundle:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>my.group.id</groupId>
    <artifactId>branding</artifactId>
    <packaging>bundle</packaging>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-DocURL>http://felix.apache.org/
site/apache-karaf.html</Bundle-DocURL>

<Fragment-Host>org.apache.karaf.webconsole.console;bundle-version="[3,4)"</
                        <Export-Package>!*</Export-Package>
                        <Import-Package>
                            javax.servlet;version=2.4,
                            javax.servlet.http;version=2.4,
                            !org.apache.felix.webconsole*,
```

```
                            org.apache.aries.blueprint,
                            org.osgi.service.blueprint.container,
                            org.osgi.service.blueprint.reflect,
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

With the `webconsole` feature installed, you can install this bundle (using `bundle:install` or by editing the `etc/startup.properties`), you will see the WebConsole with your branding.

Extending

Apache Karaf is a very flexible container that you can extend very easily.

CONSOLE

In this section, you will see how to extend the console by adding your own command.

We will leverage Apache Maven to create and build the OSGi bundle.
This OSGi bundle will use Blueprint. We don't cover the details of OSGi bundle and Blueprint, see the specific
sections for details.

Create the Maven project

To create the Maven project, we can:

- use a Maven archetype
- create by hand

Using archetype

The Maven Quickstart archetype can create an empty Maven project where you can put your project definition.

You can directly use:

```
mvn archetype:create \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=org.apache.karaf.shell.samples \
  -DartifactId=shell-sample-commands \
  -Dversion=1.0-SNAPSHOT
```

It results to a ready to use project, including a `pom.xml`.

You can also use Maven archetype in interactive mode. You will have to answer to some questions used to generate
the project with the `pom.xml`:

```
mvn archetype:generate
Choose a number:  (1/2/3/4/5/6/7/.../32/33/34/35/36) 15: : 15
Define value for groupId: : org.apache.karaf.shell.samples
Define value for artifactId: : shell-sample-commands
```

```
Define value for version:  1.0-SNAPSHOT: :
Define value for package: : org.apache.karaf.shell.samples
```

By hand

Alternatively, you can simply create the directory `shell-sample-commands` and create the `pom.xml` file inside it:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.apache.karaf.shell.samples</groupId>
  <artifactId>shell-sample-commands<artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>shell-sample-commmands</name>


  <dependencies>
    <dependency>
      <groupId>org.apache.karaf.shell</groupId>
      <artifactId>org.apache.karaf.shell.console</artifactId>
      <version>4.0.3</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>3.0.0</version>
        <configuration>
```

```
          <instructions>
            <Import-Package>
              org.apache.felix.service.command,
              org.apache.karaf.shell.commands,
              org.apache.karaf.shell.console,
              *
            </Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

Configuring for Java 6/7

We are using annotations to define commands, so we need to ensure Maven will actually use JDK 1.6 or 1.7 to compile the jar.
Just add the following snippet after the `dependencies` section.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <target>1.6</target>
        <source>1.6</source>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Loading the project in your IDE

We can use Maven to generate the needed files for your IDE:

Inside the project, run the following command

```
mvn eclipse:eclipse
```

or

```
mvn idea:idea
```

The project files for your IDE should now be created. Just open the IDE and load the project.

Creating a basic command class

We can now create the command class `HelloShellCommand.java`

```
package org.apache.karaf.shell.samples;

import org.apache.karaf.shell.api.action.Action;
import org.apache.karaf.shell.api.action.Command;
import org.apache.karaf.shell.api.action.lifecycle.Service;

@Command(scope = "test", name = "hello", description="Says
hello")
@Service
public class HelloShellCommand implements Action {

    @Override
    public Object execute() throws Exception {
        System.out.println("Executing Hello command");
        return null;
    }
}
```

Manifest

In order for Karaf to find your command, you need to add the `Karaf-Commands=*` manifest header.

This is usually done by modifying the maven bundle plugin configuration

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <configuration>
        <instructions>
            <Karaf-Commands>*</Karaf-Commands>
        </instructions>
    </configuration>
</plugin>
```

Compile

Let's try to build the jar. Remove the test classes and sample classes if you used the artifact, then from the command line, run:

```
mvn install
```

The end of the maven output should look like:

```
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
```

Test

Launch Apache Karaf and install your bundle:

```
karaf@root()> bundle:install -s
mvn:org.apache.karaf.shell.samples/shell-sample-commands/
1.0-SNAPSHOT
```

Let's try running the command:

```
karaf@root()> test:hello
Executing Hello command
```

Command completer

A completer allows you to automatically complete a command argument using <tab>. A completer is simply a bean which is
injected to a command.

Of course to be able to complete it, the command should require an argument.

Command argument

We add an argument to the HelloCommand:

```
package org.apache.karaf.shell.samples;

import org.apache.karaf.shell.api.action.Action;
import org.apache.karaf.shell.api.action.Argument;
import org.apache.karaf.shell.api.action.Command;
import org.apache.karaf.shell.api.action.Completion;
```

```
import org.apache.karaf.shell.api.action.lifecycle.Service;

@Command(scope = "test", name = "hello", description="Says
hello")
@Service
public class HelloShellCommand implements Action {

    @Argument(index = 0, name = "name", description = "The name
that sends the greet.", required = true, multiValued = false)
    @Completion(SimpleNameCompleter.class)
    String name = null;

    @Override
    public Object execute() throws Exception {
        System.out.println("Hello " + name);
        return null;
    }
}
```

Completer bean

A completer is a bean which implements the Completer interface:

```
package org.apache.karaf.shell.samples;

import org.apache.karaf.shell.api.action.lifecycle.Service;
import org.apache.karaf.shell.api.console.CommandLine;
import org.apache.karaf.shell.api.console.Completer;
import org.apache.karaf.shell.api.console.Session;
import
org.apache.karaf.shell.support.completers.StringsCompleter;

/**
 * <p>
 * A very simple completer.
 * </p>
 */
@Service
public class SimpleNameCompleter implements Completer {

    public int complete(Session session, CommandLine
commandLine, List<String> candidates) {
        StringsCompleter delegate = new StringsCompleter();
```

```
        delegate.getStrings().add("Mike");
        delegate.getStrings().add("Eric");
        delegate.getStrings().add("Jenny");
        return delegate.complete(buffer, cursor, candidates);
    }

}
```

Completers for option values

Quite often your commands will not have just arguments, but also options. You can provide completers for option values.
The snippet below shows the HelloShellCommand with an option to specify what the greet message will be.

```
package org.apache.karaf.shell.samples;

import org.apache.karaf.shell.api.action.Action;
import org.apache.karaf.shell.api.action.Argument;
import org.apache.karaf.shell.api.action.Command;
import org.apache.karaf.shell.api.action.Completion;
import org.apache.karaf.shell.api.action.Option;
import org.apache.karaf.shell.api.action.lifecycle.Service;

@Command(scope = "test", name = "hello", description="Says
hello")
@Service
public class HelloShellCommand implements Action {

    @Argument(index = 0, name = "name", description = "The name
that sends the greet.", required = true, multiValued = false)
    @Completion(SimpleNameCompleter.class)
    String name = null;

    @Option(name = "-g", aliases = "--greet", description = "The
configuration pid", required = false, multiValued = false)
    @Completion(GreetCompleter.class)
    String greet = "Hello;

    @Override
    public Object execute() throws Exception {
        System.out.println(greet + " " + name);
        return null;
```

```
    }
}
```

Completers with state

Some times we want to tune the behavior of the completer depending on the commands already executed, in the current shell
or even the rest of the arguments that have been already passed to the command. Such example is the config:set-property
command which will provide auto completion for only for the properties of the pid specified by a previously issued config:edit
command or by the option --pid.

The Session object provides map like methods for storing key/value pairs and can be used to put/get the state.
The pre-parsed CommandLine objects allows you to check the previous arguments and options on the command line and to fine tune
the behavior of the Completer.
Those two objects are given to the Completer when calling the `complete` method.

Test

Launch a Karaf instance and run the following command to install the newly created bundle:

```
karaf@root()> bundle:install -s
mvn:org.apache.karaf.shell.samples/shell-sample-commands/
1.0-SNAPSHOT
```

Let's try running the command:

```
karaf@root> test:hello <tab>
 one    two    three
```

WEBCONSOLE

You can also extend the Apache Karaf WebConsole by providing and installing a webconsole plugin.

A plugin is an OSGi bundle that register a Servlet as an OSGi service with some webconsole properties.

Using the karaf-maven-plugin

The Karaf Maven plugin allows you:

* to work with Karaf features: validate a features descriptor, add features bundle into a repository, create a KAR archive from a features descriptor, etc.
* to create Karaf commands help: it generates help from Karaf commands
* to modify Karaf instances and create distributions

PACKAGINGS

The most generally useful features of the karaf-maven-plugin are exposed as packagings. To use the packagings the pom or an ancestor must configure the karaf-maven-plugin with extensions:

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <version>4.0.3</version>
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.karaf.tooling</groupId>
            <artifactId>karaf-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Then specify the packaging in your project as usual, e.g.

```
<packaging>kar</packaging>
```

Packaging	Description

feature	The feature packaging generates a features.xml descriptor using the `karaf:features-generate-descriptor`
kar	The kar packaging generates a features.xml descriptor using the `karaf:features-generate-descriptor` and then packages a kar using the `karaf:features-create-kar`
`karaf-assembly`	Assembles a Karaf server based on the features descriptors and kar files listed as Maven dependencies.

COMMANDS GOALS

The `karaf-maven-plugin` is able to generate documentation for Karaf commands:

Goal	Description
`karaf:commands-generate-help`	Generates help for Karaf commands.

FEATURES GOALS

Normally you should use the features or kar packagings instead of these individual goals.
The `karaf-maven-plugin` provides several goals to help you create and validate features XML descriptors as well as leverage your features to create a custom Karaf distribution.

Goal	Description
`karaf:features-generate-descriptor`	Generates a features XML descriptor for a set of bundles. Used in feature and kar packagings.
`karaf:features-validate-descriptor`	Validate a features XML descriptor by checking if all the required imports can be matched to exports
`karaf:kar`	Assemble a KAR archive from a features XML descriptor. Used in kar packaging.

INSTANCES AND DISTRIBUTIONS GOALS

Normally you should use the karaf-assembly packaging instead of this individual goal.
The `karaf-maven-plugin` helps you to build custom Karaf distributions or archives existing Karaf instances:

Goal	Description

`karaf:assembly`	Assemble a server from Maven feature-repo and kar dependencies. Used in karaf-assembly packaging. See `karaf-assembly`.
`karaf:archive`	Package a server archive from an assembled server. . Used in karaf-assembly packaging. See also `karaf-assembly`.
`karaf:features-add-to-repository`	(old style manual assemblies) Copies all the bundles required for a given set of features into a directory (e.g. for creating your own Karaf-based distribution)

GOAL `KARAF:FEATURES-ADD-TO-REPOSITORY`

Consider using the karaf-assembly packaging which makes it easy to assemble a custom distribution in one step instead of this individual goal.

The `karaf:features-add-to-repository` goal adds all the required bundles for a given set of features into directory. You can use this goal to create a `/system` directory for building your own Karaf-based distribution.

By default, the Karaf core features descriptors (standard and enterprise) are automatically included in the descriptors set.

Example

The example below copies the bundles for the `spring` and `war` features defined in the Karaf features XML descriptor into the `target/features-repo` directory.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>

        <executions>
          <execution>
            <id>features-add-to-repo</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>features-add-to-repository</goal>
            </goals>
            <configuration>
              <descriptors>
```

```
                <descriptor>mvn:org.apache.karaf.features/standard/
4.0.0/xml/features</descriptor>
                <descriptor>mvn:my.groupid/my.artifactid/1.0.0/xml/
features</descriptor>
              </descriptors>
              <features>
                <feature>spring</feature>
                <feature>war</feature>
                <feature>my</feature>
              </features>
              <repository>target/features-repo</repository>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`descriptors`	`String[]`	List of features XML descriptors where the features are defined
`features`	`String[]`	List of features that bundles should be copied to the repository directory
`repository`	`File`	The directory where the bundles will be copied by the plugin goal

GOAL `KARAF:FEATURES-GENERATE-DESCRIPTOR`

Except in unusual circumstances, use the <packaging>feature</packaging> to run this goal.

The `karaf:features-generate-descriptor` goal generates a features XML file based on the Maven dependencies. By default, it will follow Maven transitive dependencies, stopping when it encounters bundles already present in features that are Maven dependencies.

A record of the dependency tree search can be found in target/history/treeListing.txt.

You can track dependency changes and warn or fail on change.

CONFIGURATION

Specify the packaging as a top level element

```
<packaging>feature</packaging>
```

You can supply a feature descriptor to extend in src/main/feature/feature.xml.

Parameter Name	Type	Description
aggregateFeatures	boolean (false)	Specifies processing of feature repositories that are (transitive) Maven dependencies. If false, all features in these repositories become dependencies of the generated feature. If true, all features in these repositories are copied into the generated feature repository.
startLevel	int	The start level for the bundles determined from Maven dependencies. This can be overridden by specifying the bundle in the source feature.xml with the desired startlevel.
includeTransitiveDependency	boolean (true)	Whether to follow Maven transitive dependencies.
checkDependencyChange	boolean (false)	Whether to record dependencies in src/main/history.dependencies.xml for change tracking.
warnOnDependencyChange	boolean (false)	whether to fail on changed dependencies (false, default) or warn in the build output (true).
logDependencyChanges	boolean (false)	If true, added and removed dependencies are shown in target/history.
overwriteChangedDependencies	boolean (false)	If true, the src/main/history/dependencies.xml file will be overwritten if it has changed.

Example

```
<project>
...
  <packaging>feature</packaging>
  <dependencies>
    <dependency>
      <groupId>org.apache</groupId>
      <artifactId>bundle1</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>
        <extensions>true</extensions>
          <configuration>
            <startLevel>80</startLevel>
            <aggregateFeatures>true</aggregateFeatures>
          </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

GOAL `KARAF:FEATURES-VALIDATE-DESCRIPTOR`

The `karaf:features-validate-descriptor` goal validates a features XML descriptor by checking if all the required imports for the bundles defined in the features can be matched to a provided export.

By default, the plugin tries to add the Karaf core features (standard and enterprise) in the repositories set.
It means that it's not required to explicitly define the Karaf features descriptor in the repository section of your features descriptor.

Example

The example below validates the features defined in the `target/features.xml` by checking all the imports and exports. It reads the definition for the packages that are exported by the system bundle from the `src/main/resources/config.properties` file.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>
          <executions>
            <execution>
              <id>validate</id>
              <phase>process-resources</phase>
              <goals>
                <goal>features-validate-descriptor</goal>
              </goals>
              <configuration>
                <file>target/features.xml</file>
                <karafConfig>src/main/resources/
config.properties</karafConfig>
              </configuration>
            </execution>
          </executions>
          <dependencies>
            <dependency>
              <groupId>org.slf4j</groupId>
              <artifactId>slf4j-simple</artifactId>
              <version>1.4.3</version>
                </dependency>
          </dependencies>
        </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`file`	`File`	The features XML descriptor file to validate. Default value: `/home/jbonofre/Workspace/karaf/manual/target/classes/features.xml`
`karafConfig`	`String`	The Karaf `config.properties` file to use during the validation process Default value: `config.properties`

`jreVersion`	`String`	The JRE version that is used during the validation process Default value: {{jre-1.5})
`karafVersion`	`String`	The target Karaf version used to get the Karaf core features (standard and enterprise) Default is the version of the plugin
`repositories`	`String[]`	Additional features XML descriptors that will be used during the validation process

GOAL `KARAF:KAR`

Except in unusual circumstances, use the <packaging>kar</packaging> to run this goal.

The `karaf:kar` goal assembles a KAR archive from a features XML descriptor file, normally generated in the same project with the `karaf:features-generate-descriptor` mojo.

KAR LAYOUT

There are two important directories in a kar:
repository/ contains a Maven structured repository of artifacts to be copied into the Karaf repository. The features descriptor and all the bundles mentioned in it are installed in this directory.
resources/ contains other resources to be copied over the Karaf installation.

Everything in target/classes is copied into the kar. Therefore resources you want installed into Karaf need to be in e.g. src/main/resources/resources. This choice is so other resources such as legal files from the maven-remote-resources-plugin can be included under META-INF in the kar, without getting installed into Karaf.

Example

```
<project>
...
  <packaging>kar</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>
```

```
        <extensions>true</extensions>
        <!-- There is no useful configuration for the kar mojo. The
features-generate-descriptor mojo configuration may be useful -->
      </plugin>
    </plugins>
  </build>
</project>
```

GOAL KARAF:COMMANDS-GENERATE-HELP

The `karaf:commands-generate-help` goal generates documentation containing Karaf commands help.

It looks for Karaf commands in the current project class loader and generates the help as displayed with the --help
option in the Karaf shell console.

Example

The example below generates help for the commands in the current project:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>

        <executions>
          <execution>
            <id>document-commands</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>commands-generate-help</goal>
            </goals>
            <configuration>
              <targetFolder>/home/jbonofre/Workspace/karaf/manual/
target/docbook/sources</targetFolder>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
```

```
  </build>
</project>
```

Parameters

Name	Type	Description
`targetFolder`	`File`	The directory where the documentation output files are to be generated Default value: /home/jbonofre/Workspace/karaf/manual/target/docbkx/sources
`format`	`String`	The output format (docbx or conf) of the commands documentation. Default value: docbx
`classLoader`	`String`	The class loader to use in loading the commands. Default value: $

GOAL `KARAF:ARCHIVE`

Normally this is run as part of the karaf-assembly packaging.

The `karaf:archive` goal packages a Karaf instance archive from a given assembled instance.

Both tar.gz and zip formats are generated in the destination folder.

Example

The example below create archives for the given Karaf instance:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.0.3</version>
          <executions>
            <execution>
              <id>generate</id>
              <phase>package</phase>
              <goals>
```

```
              <goal>archive</goal>
            </goals>
            <configuration>
              <destDir>/home/jbonofre/Workspace/karaf/manual/
target</destDir>
              <targetServerDirectory>/home/jbonofre/Workspace/
karaf/manual/target/assembly</targetServerDirectory>
              <targetFile>/home/jbonofre/Workspace/karaf/manual/
pom.xml</targetFile>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`destDir`	`File`	The target directory of the project. Default value: /home/jbonofre/Workspace/karaf/manual/target
`targetServerDirectory`	`File`	The location of the server repository. Default value: /home/jbonofre/Workspace/karaf/manual/target/assembly
`targetFile`	`File`	The target file to set as the project's artifact. Default value: /home/jbonofre/Workspace/karaf/manual/pom.xml

Custom distributions

As Karaf is an OSGi container, it's heavily used as as application and middleware kernel.

You may wish to construct your own Karaf distribution preconfigured to your requirements.
This custom distribution could contain:

- branding to change the Karaf console look-and-feel
- configuration files (in the etc folder) altered to your requirements
- pre-packaged artifacts in the deploy folder
- a pre-populated system repository (containing your own bundle and features descriptor)
- renamed or specific scripts in the bin folder
- system documentation files

MAVEN ASSEMBLY

The recommended way to create a Karaf server assembly is to use the karaf-assembly packaging with the karaf-maven-plugin. This assembles a server from the maven dependencies in the project pom. After explanation of the configuration options we present an example. The Karaf project effectively uses this packaging to assemble the official Karaf distributions, but due to maven limitations we have to simulate rather than use the karaf-assembly packaging.

This packaging creates tar.gz and zip archives containing the assembled server. They are identical except that zip archives don't unpack with appropriate unix file permissions for the scripts.

Maven dependencies

Maven dependencies in a karaf-assembly project can be feature repositories (classifier "features") or kar archives. Feature repositories are installed in the internal "system" Maven structured repository. Kar archives have their content unpacked on top of the server as well as contained feature repositories installed.

The Maven scope of a dependency determines whether its feature repository is listed in the features service configuration file etc/org.apache.karaf.features.cfg featuresRepositories property:

compile (default): All the features in the repository (or for a kar repositories) will be installed into the startup.properties. The feature repo is not listed in the features service configuration file.

runtime: feature installation is controlled by <startupFeature>, <bootFeature>, and <installedFeature> elements in the karaf-maven-plugin configuration. The feature repo uri is listed in the features service configuration file.

Plugin configuration

Control how features are installed using these elements referring to features from installed feature repositories:

- <startupFeature>foo</startupFeature> This will result in the feature bundles being listed in startup.properties at the appropriate start level and the bundles being copied into the "system" internal repository. You can use feature_name or feature_name/feature_version formats.
- <bootFeature>bar</bootFeature> This will result in the feature name added to boot-features in the features service configuration file and all the bundles in the feature copied into the "system" internal repository. You can use feature_name or feature_name/feature_version formats.
- <installedFeature>baz</installedFeature> This will result in all the bundles in the feature being installed in the "system" internal repository. Therefore at runtime the feature may be installed without access to external repositories. You can use feature_name or feature_name/feature_version formats.

Minimal Distribution Example

This is the minimal assembly pom changed to use the packaging and annotated

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
...
    </parent>

    <groupId>org.apache.karaf</groupId>
    <artifactId>apache-karaf-minimal</artifactId>
    <version>4.0.3</version>
    <packaging>karaf-assembly</packaging>
    <name>Apache Karaf :: Assemblies :: Minimal
Distribution</name>

    <dependencies>
```

```
        <dependency>
        <!-- scope is compile so all features (there is only
one) are installed into startup.properties and the feature repo
itself is not added in etc/org.apache.karaf.features.cfg file -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>framework</artifactId>
            <version>4.0.3</version>
            <type>kar</type>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in
etc/org.apache.karaf.features.cfg file, and features will
installed into the system directory -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>standard</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <!-- if you want to include resources in the
distribution -->
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>false</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/
filtered-resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>

        <plugins>
```

```
            <!-- if you want to include resources in the
distribution -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <version>2.6</version>
                <executions>
                    <execution>
                        <id>process-resources</id>
                        <goals>
                            <goal>resources</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <!-- karaf-maven-plugin will call both assembly and
archive goals -->
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <!-- no startupFeatures -->
                    <bootFeatures>
                        <feature>standard</feature>
                        <feature>management</feature>
                    </bootFeatures>
                    <!-- no installedFeatures -->
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Custom Distribution Example

It's possible to specify feature versions using the name/version format.

For instance, to pre-install Spring 4.0.7.RELEASE_1 feature in your custom distribution, you can use the following pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>my.custom</groupId>
    <artifactId>my.distribution</artifactId>
    <version>1.0</version>
    <packaging>karaf-assembly</packaging>

    <dependencies>
        <dependency>
        <!-- scope is compile so all features (there is only
one) are installed into startup.properties and the feature repo
itself is not added in etc/org.apache.karaf.features.cfg file -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>framework</artifactId>
            <version>4.0.0</version>
            <type>kar</type>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in
etc/org.apache.karaf.features.cfg file, and features will
installed into the system directory if specify in the plugin
configuration -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>standard</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in
etc/org.apache.karaf.features.cfg file, and features will
installed into the system directory if specify in the plugin
configuration -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>spring</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

```
    <build>
        <!-- if you want to include resources in the
distribution -->
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>false</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/
filtered-resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>

        <plugins>
            <!-- if you want to include resources in the
distribution -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <version>2.6</version>
                <executions>
                    <execution>
                        <id>process-resources</id>
                        <goals>
                            <goal>resources</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <version>4.0.0</version>
                <extensions>true</extensions>
```

```
                <configuration>
                    <!-- no startupFeatures -->
                    <bootFeatures>
                        <feature>jaas</feature>
                        <feature>shell</feature>
                        <feature>ssh</feature>
                        <feature>management</feature>
                        <feature>bundle</feature>
                        <feature>config</feature>
                        <feature>deployer</feature>
                        <feature>diagnostic</feature>
                        <feature>instance</feature>
                        <feature>kar</feature>
                        <feature>log</feature>
                        <feature>package</feature>
                        <feature>service</feature>
                        <feature>system</feature>
                    </bootFeatures>
                    <installedFeatures>
                        <feature>wrapper</feature>
                        <feature>spring/4.0.7.RELEASE_1</feature>
                    </installedFeatures>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

(DEPRECATED OLD STYLE) MAVEN ASSEMBLY

Basically a Karaf custom distribution involves:
1. Uncompressing a standard Karaf distribution in a given directory.
2. Populating the system repo with your features.
3. Populating the lib directory with your branding or other system bundle jar files.
4. Overriding the configuration files in the etc folder.

These tasks could be performed using scripting, or more easily and portable, using Apache Maven and a set of Maven plugins.

For instance, the Maven POM could look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM
```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <groupId>my.company</groupId>
  <artifactId>mycustom-karaf</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <name>My Unix Custom Karaf Distribution</name>

  <properties>
    <karaf.version>4.0.3</karaf.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.karaf</groupId>
      <artifactId>apache-karaf</artifactId>
      <version>4.0.3</version>
      <type>tar.gz</type>
    </dependency>
    <dependency>
      <groupId>org.apache.karaf</groupId>
      <artifactId>apache-karaf</artifactId>
      <version>4.0.3</version>
      <type>xml</type>
      <classifier>features</classifier>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>/home/jbonofre/Workspace/karaf/manual/src/
main/filtered-resources</directory>
        <filtering>true</filtering>
        <includes>
          <include>**/*</include>
        </includes>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
```

```
        <executions>
          <execution>
            <id>filter</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>resources</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>4.0.3</version>
        <executions>
          <execution>
           <id>add-features-to-repo</id>
           <phase>generate-resources</phase>
           <goals>
             <goal>add-features-to-repo</goal>
           </goals>
           <configuration>
              <descriptors>
                <descriptor>mvn:org.apache.karaf/apache-karaf/
4.0.3/xml/features</descriptor>
                <descriptor>file:/home/jbonofre/Workspace/karaf/
manual/target/classes/my-features.xml</descriptor>
              </descriptors>
              <features>
                <feature>my-feature</feature>
              </features>
           </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy</id>
            <phase>generate-resources</phase>
            <goals>
```

```
              <goal>copy</goal>
            </goals>
            <configuration>
               <!-- Define here the artifacts which should be
considered in the assembly -->
               <!-- For instance, the branding jar -->
               <artifactItems>
                 <artifactItem>
                    <groupId>my.groupId</groupId>
                    <artifactId>my.branding.id</artifactId>
                    <version>1.0</version>
                    <outputDirectory>target/
dependencies</outputDirectory>
                    <destFileName>mybranding.jar</destFileName>
                 </artifactItem>
               </artifactItems>
            </configuration>
          </execution>
          <execution>
            <!-- Uncompress the standard Karaf distribution -->
            <id>unpack</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>unpack</goal>
            </goals>
            <configuration>
              <artifactItems>
                <artifactItem>
                  <groupId>org.apache.karaf</groupId>
                  <artifactId>apache-karaf</artifactId>
                  <type>tar.gz</type>
                  <outputDirectory>target/
dependencies</outputDirectory>
                </artifactItem>
              </artifactItems>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
        <executions>
```

```
          <execution>
            <id>bin</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptors>
                <descriptor>src/main/descriptors/
bin.xml</descriptor>
              </descriptors>
              <appendAssemblyId>false</appendAssemblyId>
              <tarLongFileMode>gnu</tarLongFileMode>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

</project>
```

The Maven POM will download the Karaf standard distribution and prepare resources to be processed by the Maven assembly plugin.

Your Maven project structure should look like:

- pom.xml: the previous POM file
- src/main/descriptors/bin.xml: the assembly Maven plugin descriptor (see below)
- src/main/filtered-resources: contains all resource files that have Maven property values to be filtered/replaced. Typically, this will include features descriptor and configuration files.
- src/main/distribution: contains all raw files which will be copied as-is into your custom distribution.

For instance, `src/main/filtered-resources` could contain:

- `my-features.xml` where Maven properties will be replaced
- `etc/org.apache.karaf.features.cfg` file containing your my-features descriptor:

```
#
# Comma separated list of features repositories to
register by default
#
featuresRepositories=mvn:org.apache.karaf/apache-karaf/
```

```
4.0.3/xml/features,mvn:my.groupId/my-features/4.0.3/xml/
features

#
# Comma separated list of features to install at startup
#
featuresBoot=config,ssh,management,my-feature
```

The `src/main/distribution` contains all your custom Karaf configuration files and script, as, for examples:

- etc/org.ops4j.pax.logging.cfg

```
# Root logger
log4j.rootLogger=INFO, out, osgi:VmLogAppender
log4j.throwableRenderer=org.apache.log4j.OsgiThrowableRenderer

# CONSOLE appender not used by default
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
| %-5.5p | %-16.16t | %-32.32C %4L | %X{bundle.id} -
%X{bundle.name} - %X{bundle.version} | %m%n

# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ABSOLUTE}
| %-5.5p | %-16.16t | %-32.32C %4L | %X{bundle.id} -
%X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.home}/log/
my-customer-distribution.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10

# Sift appender
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=my-custom
log4j.appender.sift.appender=org.apache.log4j.FileAppender
log4j.appender.sift.appender.layout=org.apache.log4j.PatternLayout
log4j.appender.sift.appender.layout.ConversionPattern=%d{ABSOLUTE}
| %-5.5p | %-16.16t | %-32.32c{1} | %-32.32C %4L | %m%n
```

```
log4j.appender.sift.appender.file=${karaf.data}/log/$\\{bundle.name\
log4j.appender.sift.appender.append=true
```

- etc/system.properties

```
#
# The properties defined in this file will be made
available through system
# properties at the very beginning of the FAS boot
process.
#

# Log level when the pax-logging service is not available
# This level will only be used while the pax-logging
service bundle
# is not fully available.
# To change log levels, please refer to the
org.ops4j.pax.logging.cfg file
# instead.
org.ops4j.pax.logging.DefaultServiceLog.level=ERROR

#
# Name of this custom instance.
#
karaf.name=my-custom

#
# Default repository where bundles will be loaded from
before using
# other Maven repositories. For the full Maven
configuration, see the
# org.ops4j.pax.url.mvn.cfg file.
#
karaf.default.repository=system

#
# Location of a shell script that will be run when
starting a shell
# session. This script can be used to create aliases and
define
# additional commands.
#
karaf.shell.init.script=${karaf.home}/etc/
shell.init.script
```

```
#
# Set this empty property to avoid errors when validating
xml documents.
#
xml.catalog.files=

#
# Suppress the bell in the console when hitting backspace
to many times
# for example
#
jline.nobell=true

#
# Default port for the OSGi HTTP Service
#
org.osgi.service.http.port=8181

#
# Allow usage of ${custom.home} as an alias for
${karaf.home}
#
custom.home=${karaf.home}
```

- etc/users.properties

```
admin=admin,admin
```

- You can add a `etc/custom.properties`, it's a placeholder for any override you may need. For instance:

```
karaf.systemBundlesStartLevel=50
obr.repository.url=http://svn.apache.org/repos/asf/
servicemix/smx4/obr-repo/repository.xml
org.osgi.framework.system.packages.extra = \
  org.apache.karaf.branding; \
  com.sun.org.apache.xalan.internal.xsltc.trax; \
  com.sun.org.apache.xerces.internal.dom; \
  com.sun.org.apache.xerces.internal.jaxp; \
  com.sun.org.apache.xerces.internal.xni; \
  com.sun.jndi.ldap
```

Now, we can "assemble" our custom distribution using the Maven assembly plugin. The Maven assembly plugin uses an
assembly descriptor, configured in POM above to be `src/main/descriptors/bin.xml`:

```
<assembly>

    <id>bin</id>

    <formats>
        <format>tar.gz</format>
    </formats>

    <fileSets>

        <!-- Expanded Karaf Standard Distribution -->
        <fileSet>
            <directory>target/dependencies/
apache-karaf-4.0.3</directory>
            <outputDirectory>/</outputDirectory>
            <excludes>
                <exclude>**/demos/**</exclude>
                <exclude>bin/**</exclude>
                <exclude>etc/system.properties</exclude>
                <exclude>etc/users.properties</exclude>
                <exclude>etc/
org.apache.karaf.features.cfg</exclude>
                <exclude>etc/org.ops4j.pax.logging.cfg</exclude>
                <exclude>LICENSE</exclude>
                <exclude>NOTICE</exclude>
                <exclude>README</exclude>
                <exclude>RELEASE-NOTES</exclude>
                <exclude>karaf-manual*.html</exclude>
                <exclude>karaf-manual*.pdf</exclude>
            </excludes>
        </fileSet>

        <!-- Copy over bin/* separately to get the correct file
mode -->
        <fileSet>
            <directory>target/dependencies/
apache-karaf-4.0.3</directory>
            <outputDirectory>/</outputDirectory>
```

```
            <includes>
                <include>bin/admin</include>
                <include>bin/karaf</include>
                <include>bin/start</include>
                <include>bin/stop</include>
            </includes>
            <fileMode>0755</fileMode>
        </fileSet>

        <!-- Copy over jar files -->
        <fileSet>
            <directory>target/dependencies</directory>
            <includes>
                <include>my-custom.jar</include>
            </includes>
            <outputDirectory>/lib/</outputDirectory>
        </fileSet>

        <fileSet>
            <directory>src/main/distribution</directory>
            <outputDirectory>/</outputDirectory>
            <fileMode>0644</fileMode>
        </fileSet>
        <fileSet>
            <directory>target/classes/etc</directory>
            <outputDirectory>/etc/</outputDirectory>
            <lineEnding>unix</lineEnding>
            <fileMode>0644</fileMode>
        </fileSet>

        <fileSet>
            <directory>target/features-repo</directory>
            <outputDirectory>/system</outputDirectory>
        </fileSet>

    </fileSets>

    <files>
        <file>
            <source>/home/jbonofre/Workspace/karaf/manual/target/
dependencies/apache-karaf-4.0.3/bin/karaf</source>
            <outputDirectory>/bin/</outputDirectory>
            <destName>my-custom</destName>
```

```
            <fileMode>0755</fileMode>
            <lineEnding>unix</lineEnding>
        </file>
        <file>
            <source>/home/jbonofre/Workspace/karaf/manual/target/
classes/features.xml</source>
            <outputDirectory>/system/my.groupid/my-features/
4.0.3</outputDirectory>
            <destName>my-features-4.0.3-features.xml</destName>
            <fileMode>0644</fileMode>
            <lineEnding>unix</lineEnding>
        </file>
    </files>

</assembly>
```

To build your custom Karaf distribution, just run:

```
mvn install
```

You will find your Karaf custom distribution tar.gz in the target directory.

ROADMAP

A distribution goal is in preparation in the next Karaf

CUSTOM DISTRIBUTION EXAMPLES

- Apache ServiceMix 4
- Apache ServiceMix NMR
- BuildProcess BuildEraser

OSGi services

Basic bundle creation using the Felix maven-bundle-plugin

Create a bundle instead of a normal jar by using a pom file like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>my.groupId</groupId>
    <artifactId>my.bundle</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>My Bundle</name>
    <description>My bundle short description</description>

    <build>
        <resources>
            <resource>
                <directory>/home/jbonofre/Workspace/karaf/manual/
src/main/resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
```

```
<Bundle-SymbolicName>manual</Bundle-SymbolicName>
                        ...
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

ADD EXTENDED INFORMATION TO BUNDLES

Karaf supports a OSGI-INF/bundle.info file in a bundle. This file is extended description of the bundle. It supports ASCII character declarations (for adding color, formatting, etc) and some simple Wiki syntax.

Simply add a src/main/resources/OSGI-INF/bundle.info file containing, for instance:

```
h1. SYNOPSIS
    The Apache Software Foundation provides support for the
Apache community of open-source software projects.
    The Apache projects are characterized by a collaborative,
consensus based development process, an open and
    pragmatic software license, and a desire to create high
quality software that leads the way in its field.
    We consider ourselves not simply a group of projects sharing
a server, but rather a community of developers
    and users.

h1. DESCRIPTION
    Long description of your bundle, including usage, etc.

h1.SEE ALSO
    [http://yourside\]
    [http://yourside/docs\]
```

You can display this extended information using:

```
root@karaf> bundles:info
```

WIKI SYNTAX

Karaf supports some simple wiki syntax in bunde info files:

```
h1., h2., ... : Headings
* : Enumerations
[http://....] : links
```

Creating bundles for non OSGi third party dependencies

DYNAMICALLY WRAPPING JARS

Karaf supports the wrap: protocol execution.

It allows for directly deploying third party dependencies, like Apache Commons Lang:

```
root@karaf> bundles:install wrap:mvn:commons-lang/commons-lang/
2.4
```

The wrap protocol creates a bundle dynamically using the bnd. Confiugurations can be added in the wrap URL:

from the shell

```
root@karaf> bundles:install 'wrap:mvn:commons-lang/commons-lang/
2.4$Bundle-SymbolicName=commons-lang&Bundle-Version=2.4'
```

from features.xml

```
<bundle>wrap:mvn:commons-lang/commons-lang/
2.4$Bundle-SymbolicName=commons-lang&undle-Version=2.4</bundle>
```

STATICALLY BUNDLING JARS

You can also create a wrap bundle for a third party dependency.
This bundle is simply a Maven POM that shades an existing jar and package into a jar bundle.

For instance, to create an OSGi bundle that wraps Apache Commons Lang, simply define the following Maven POM:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>osgi.commons-lang</groupId>
    <artifactId>osgi.commons-lang</artifactId>
    <version>2.4</version>
    <packaging>bundle</packaging>
    <name>commons-lang OSGi Bundle</name>
    <description>This OSGi bundle simply wraps
commons-lang-2.4.jar artifact.</description>

    <dependencies>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.4</version>
            <optional>true</optional>
        </dependency>
    </dependencies>

    <build>
        <defaultGoal>install</defaultGoal>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>1.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <includes>

<include>commons-lang:commons-lang</include>
                            </includes>
                        </artifactSet>
                        <filters>
                            <filter>

<artifact>commons-lang:commons-lang</artifact>
                                <excludes>
                                    <exclude>**</exclude>
```

```
                                    </excludes>
                                </filter>
                            </filters>

<promoteTransitiveDependencies>true</promoteTransitiveDependencies>

<createDependencyReducedPom>true</createDependencyReducedPom>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.1.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>

<Bundle-SymbolicName>manual</Bundle-SymbolicName>
                        <Export-Package></Export-Package>
                        <Import-Package></Import-Package>

<_versionpolicy>[$(version;==;$(@)),$(version;+;$(@)))</_versionpolicy>

<_removeheaders>Ignore-Package,Include-Resource,Private-Package,Embed-Deper
                    </instructions>
                    <unpackBundle>true</unpackBundle>
                </configuration>
            </plugin>
        </build>

</project>
```

The resulting OSGi bundle can now be deployed directly:

```
root@karaf> bundles:install -s mvn:osgi.commons-lang/
osgi.commons-lang/2.4
```

Some more information is available at http://gnodet.blogspot.com/2008/09/id-like-to-talk-bit-about-third-party.html, http://blog.springsource.com/2008/02/18/creating-osgi-bundles/ and http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html.

Archetypes

Karaf provides archetypes to easily create commands, manage [features or repository and create a kar archive.
This section describes each of them and explain How to use it.

CREATE A COMMAND (KARAF-COMMAND-ARCHETYPE)

This archetype creates a Maven skeleton project that you will use to develop new Karaf commands.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.karaf.archetypes \
  -DarchetypeArtifactId=karaf-command-archetype \
  -DarchetypeVersion=4.0.0 \
  -DgroupId=com.mycompany \
  -DartifactId=com.mycompany.command \
  -Dversion=1.0-SNAPSHOT \
  -Dpackage=com.mycompany.package
```

Additional parameters

During the maven creation process, additional questions will be asked on the console :

- ◦ Define value for property 'command':
  The name of the command (list, add-item, ...)
- ◦ Define value for property 'description':
  Provide a description of the command that you want to create. This description will be displayed in the Karaf console when the parameter --help is used
- ◦ Define value for property 'scope':
  This value represents the family name to which the command belongs (features, osgi, admin, jaas, ...)

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate
(default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate
(default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate
(default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one
from
[org.apache.karaf.archetypes:karaf-command-archetype:4.0.0]
found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId =
com.mycompany.command
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package =
com.mycompany.package
Define value for property 'command': : list
Define value for property 'description': : List
sample command
Define value for property 'scope': : my
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.command
version: 1.0-SNAPSHOT
package: com.mycompany.package
command: list
description: List sample command
scope: my
 Y: :
[INFO]
```

```
------------------------------------------------------------------------
[INFO] Using following parameters for creating
project from Archetype:
karaf-command-archetype:4.0.0
[INFO]
------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value:
com.mycompany.command
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value:
com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/
mycompany/package
[INFO] Parameter: package, Value:
com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: scope, Value: my
[INFO] Parameter: description, Value: List sample
command
[INFO] Parameter: command, Value: list
[INFO] Parameter: artifactId, Value:
com.mycompany.command
[WARNING] Don't override file
/com.mycompany.command/pom.xml
[INFO] project created from Archetype in dir:
/com.mycompany.command
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 27.204s
[INFO] Finished at: Mon Dec 19 09:38:49 CET 2011
[INFO] Final Memory: 7M/111M
[INFO]
------------------------------------------------------------------------
```

Next, you can import your project in Eclipse/IntelliJ and developp the karaf command.

CREATE AN OSGI BUNDLE (KARAF-BUNDLE-ARCHETYPE)

This archetype creates a Maven skeleton to create an OSGi bundle, including a bundle Activator (a special callback class for bundle start/stop).

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-bundle-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.bundle \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.package
```

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-bundle-archetype:4.0.0] found
in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.bundle
[INFO] Using property: version = 1.0-SNAPSHOT
```

```
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.bundle
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO]
----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from
Archetype: karaf-bundle-archetype:4.0.0
[INFO]
----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.bundle
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/
package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.bundle
[INFO] project created from Archetype in dir:
/com.mycompany.bundle
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 7.895s
[INFO] Finished at: Mon Dec 19 11:41:44 CET 2011
[INFO] Final Memory: 8M/111M
[INFO]
------------------------------------------------------------------------
```

CREATE AN OSGI BLUEPRINT BUNDLE (KARAF-BLUEPRINT-ARCHETYPE)

This archetype creates a Maven skeleton project to create an OSGi blueprint bundle, including a sample bean exposed as an OSGi service in the blueprint XML descriptor.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-blueprint-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.blueprint \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.blueprint
```

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-blueprint-archetype:4.0.0]
found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.blueprint
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.blueprint
version: 1.0-SNAPSHOT
package: com.mycompany.package
```

```
 Y: :
[INFO]
------------------------------------------------------------------------
[INFO] Using following parameters for creating project from
Archetype: karaf-blueprint-archetype:4.0.0
[INFO]
------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.blueprint
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/
package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.blueprint
[INFO] project created from Archetype in dir:
/com.mycompany.blueprint
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 1:06:36.741s
[INFO] Finished at: Mon Dec 19 13:04:43 CET 2011
[INFO] Final Memory: 7M/111M
[INFO]
------------------------------------------------------------------------
```

CREATE A FEATURES XML (KARAF-FEATURE-ARCHETYPE)

This archetype creates a Maven skeleton project which create a features XML file, using the dependencies that you define in the POM.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-feature-archetype \
    -DarchetypeVersion=4.0.0 \
```

```
    -DgroupId=my.company \
    -DartifactId=my.company.feature \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=my.company.package
```

Result of maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-feature-archetype:4.0.0]
found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.feature
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.feature
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO]
------------------------------------------------------------------------
[INFO] Using following parameters for creating project from
Archetype: karaf-feature-archetype:4.0.0
[INFO]
------------------------------------------------------------------------
```

```
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.feature
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/
package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.feature
[INFO] project created from Archetype in dir:
/com.mycompany.feature
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 7.262s
[INFO] Finished at: Mon Dec 19 13:20:00 CET 2011
[INFO] Final Memory: 7M/111M
[INFO]
------------------------------------------------------------------------
```

CREATE A KAR FILE (KARAF-KAR-ARCHETYPE)

This archetype creates a Maven skeleton project including a features XML sample, used to generate a KAR file.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-kar-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.kar \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.package
```

Result of maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-kar-archetype:4.0.0] found in
catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.kar
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.kar
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO]
------------------------------------------------------------------------
[INFO] Using following parameters for creating project from
Archetype: karaf-kar-archetype:4.0.0
[INFO]
------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.kar
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/
```

```
package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.kar
[INFO] project created from Archetype in dir: /com.mycompany.kar
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 7.465s
[INFO] Finished at: Mon Dec 19 13:30:15 CET 2011
[INFO] Final Memory: 8M/157M
[INFO]
------------------------------------------------------------------------
```

Security framework

Karaf supports JAAS with some enhancements to allow JAAS to work nicely in an OSGi environment. This framework also features an OSGi keystore manager with the ability to deploy new keystores or truststores at runtime.

OVERVIEW

This feature allows runtime deployment of JAAS based configuration for use in various parts of the application. This
includes the remote console login, which uses the `karaf` realm, but which is configured with a dummy login module
by default. These realms can also be used by the NMR, JBI components or the JMX server to authenticate users logging in
or sending messages into the bus.

In addition to JAAS realms, you can also deploy keystores and truststores to secure the remote shell console, setting up HTTPS connectors or using certificates for WS-Security.

A very simple XML schema for spring has been defined, allowing the deployment of a new realm or a new keystore very easily.

SCHEMA

To override or deploy a new realm, you can use the following XSD which is supported by a Spring namespace handler and can thus be defined in a Spring xml configuration file.

Following is the XML Schema to use when defining Karaf realms:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--

    Licensed to the Apache Software Foundation (ASF) under one or
more
    contributor license agreements.  See the NOTICE file
distributed with
    this work for additional information regarding copyright
ownership.
    The ASF licenses this file to You under the Apache License,
```

```
Version 2.0
    (the "License"); you may not use this file except in compliance
with
    the License.  You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing,
software
    distributed under the License is distributed on an "AS IS"
BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
    See the License for the specific language governing permissions
and
    limitations under the License.

-->
<xs:schema elementFormDefault='qualified'
           targetNamespace='http://karaf.apache.org/xmlns/jaas/
v1.1.0'
           xmlns:xs='http://www.w3.org/2001/XMLSchema'
           xmlns:bp="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:tns='http://karaf.apache.org/xmlns/jaas/v1.1.0'>

    <xs:import namespace="http://www.osgi.org/xmlns/blueprint/
v1.0.0"/>

    <xs:element name="config">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="module" minOccurs="0"
maxOccurs="unbounded">
                    <xs:complexType mixed="true">
                        <xs:attribute name="name" use="optional"
type="xs:string"/>
                        <xs:attribute name="className"
use="required" type="xs:string"/>
                        <xs:attribute name="flags"
default="required">
                            <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                    <xs:enumeration
value="required"/>
                                    <xs:enumeration
value="requisite"/>
                                    <xs:enumeration
```

```
value="sufficient"/>
                                        <xs:enumeration
value="optional"/>
                                    </xs:restriction>
                                </xs:simpleType>
                            </xs:attribute>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="name" use="required"
type="xs:string"/>
                <xs:attribute name="rank" use="optional" default="0"
type="xs:int"/>
            </xs:complexType>
        </xs:element>

        <xs:element name="keystore">
            <xs:complexType>
                <xs:attribute name="name" use="required"
type="xs:string"/>
                <xs:attribute name="rank" use="optional" default="0"
type="xs:int"/>
                <xs:attribute name="path" use="required"
type="xs:string"/>
                <xs:attribute name="keystorePassword" use="optional"
type="xs:string"/>
                <xs:attribute name="keyPasswords" use="optional"
type="xs:string"/>
            </xs:complexType>
        </xs:element>

</xs:schema>
```

You can find the schema at the following location.

Here are two examples using this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <!-- Bean to allow the ${karaf.base} property to be correctly
resolved -->
    <ext:property-placeholder placeholder-prefix="${["
```

```
placeholder-suffix="]"/>

    <jaas:config name="myrealm">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            users = $[karaf.base]/etc/users.properties
        </jaas:module>
    </jaas:config>

</blueprint>
```

```
<jaas:keystore xmlns:jaas="http://karaf.apache.org/xmlns/jaas/
v1.1.0"
               name="ks"
               rank="1"
               path="classpath:privatestore.jks"
               keystorePassword="keyStorePassword"
               keyPasswords="myalias=myAliasPassword">
</jaas:keystore>
```

The `id` attribute is the blueprint id of the bean, but it will be used by default as the name of the realm if no
`name` attribute is specified. Additional attributes on the `config` elements are a `rank`, which is an integer.
When the LoginContext looks for a realm for authenticating a given user, the realms registered in the OSGi registry are
matched against the required name. If more than one realm is found, the one with the highest rank will be used, thus
allowing the override of some realms with new values. The last attribute is `publish` which can be set to false to
not publish the realm in the OSGi registry, thereby disabling the use of this realm.

Each realm can contain one or more module definitions. Each module identifies a LoginModule and the `className`
attribute must be set to the class name of the login module to use. Note that this login module must be available from
the bundle classloader, so either it has to be defined in the bundle itself, or the needed package needs to be correctly
imported. The `flags` attribute can take one of four values that are explained on the JAAS documentation.
The content of the `module` element is parsed as a properties file and will be used to further configure the login module.

Deploying such a code will lead to a JaasRealm object in the OSGi registry, which will then be used when using the JAAS login module.

Configuration override and use of the `rank` attribute

The `rank` attribute on the `config` element is tied to the ranking of the underlying OSGi service. When the JAAS
framework performs an authentication, it will use the realm name to find a matching JAAS configuration. If multiple
configurations are used, the one with the highest `rank` attribute will be used.
So if you want to override the default security configuration in Karaf (which is used by the ssh shell, web console and
JMX layer), you need to deploy a JAAS configuration with the name `name="karaf"` and `rank="1"`.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <!-- Bean to allow the $[karaf.base] property to be correctly
resolved -->
    <ext:property-placeholder placeholder-prefix="$["
placeholder-suffix="]"/>

    <jaas:config name="karaf" rank="1">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            users = $[karaf.base]/etc/users.properties
            ...
        </jaas:module>
    </jaas:config>

</blueprint>
```

ARCHITECTURE

Due to constraints in the JAAS specification, one class has to be available for all bundles.
This class is called ProxyLoginModule and is a LoginModule that acts as a proxy for an OSGi defines LoginModule. If you plan to integrate this feature into another OSGi

runtime, this class must be made available from the system classloader and the related package be part of the boot delegation classpath (or be deployed as a fragment attached to the system bundle).

The xml schema defined above allows the use of a simple xml (leveraging spring xml extensibility) to configure and
register a JAAS configuration for a given realm. This configuration will be made available into the OSGi registry as a
JaasRealm and the OSGi specific Configuration will look for such services.
Then the proxy login module will be able to use the information provided by the realm to actually load the class from
the bundle containing the real login module.

Karaf itself provides a set of login modules ready to use, depending of the authentication backend that you need.

In addition of the login modules, Karaf also support backend engine. The backend engine is coupled to a login module and
allows you to manipulate users and roles directly from Karaf (adding a new user, delete an existing user, etc).
The backend engine is constructed by a backend engine factory, registered as an OSGi service.
Some login modules (for security reason for instance) don't provide backend engine.

AVAILABLE REALM AND LOGIN MODULES

Karaf comes with a default realm named "karaf" using login modules.

Karaf also provides a set of login modules and backend engines to handle authentication needs for your environment.

PropertiesLoginModule

LoginModule	org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
BackendEngineFactory	org.apache.karaf.jaas.modules.properties.PropertiesBackendEngineFactor

This login module is the one configured by default. It uses a properties text file to load the users, passwords and roles.

Name	Description
`users`	location of the properties file

This file uses the properties file format.
The format of the properties is as follows, with each line defining a user, its password and associated roles:

```
user=password[,role][,role]...
```

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = ${karaf.etc}/users.properties
    </jaas:module>
</jaas:config>
```

The PropertiesLoginModule provides a backend engine allowing:

- add a new user
- delete an existing user
- list the users, groups, and roles
- add a new role to an user
- delete a role from an user
- add an user into a group
- remove an user from a group
- add a role to a group
- delete a role from a group

To enable the backend engine, you have to register the corresponding OSGi service. For instance, the following blueprint
shows how to register the PropertiesLoginModule and the corresponding backend engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <jaas:config name="karaf" rank="-1">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            users = ${karaf.etc}/users.properties
        </jaas:module>
    </jaas:config>

    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory"/>
    </service>

</blueprint>
```

OsgiConfigLoginModule

LoginModule	org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule
BackendEngineFactory	N/A

The OsgiConfigLoginModule uses the OSGi ConfigurationAdmin service to provide the users, passwords and roles.

Name	Description
`pid`	the PID of the configuration containing user definitions

The format of the configuration is the same than for the `PropertiesLoginModule` with properties prefixed with `user.`.

For instance, in the Karaf etc folder, we create a file `org.apache.karaf.authentication.cfg` containing:

```
user.karaf=karaf,admin
user.user=password,role
```

The following blueprint shows how to use this configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0">

    <jaas:config name="karaf" rank="-1">
        <jaas:module
classNameName="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule"
                     flags="required">
            pid = org.apache.karaf.authentication
        </jaas:module>
    </jaas:config>

</blueprint>
```

NB: the OsgiConfigLoginModule doesn't provide a backend engine.

JDBCLoginModule

LoginModule	org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule
BackendEngineFactory	org.apache.karaf.jaas.modules.jdbc.JDBCBackendEngineFactory

The JDBCLoginModule uses a database to load the users, passwords and roles from a provided data source *(normal or XA)*.

The data source and the queries for password and role retrieval are configurable using the following parameters.

Name	Description
`datasource`	The datasource as on OSGi ldap filter or as JDNI name
`query.password`	The SQL query that retries the password of the user
`query.role`	The SQL query that retries the roles of the user

Passing a data source as an OSGi ldap filter

To use an OSGi ldap filter, the prefix osgi: needs to be provided, as shown below:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                 flags="required">
        datasource =
osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
        query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
        query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
    </jaas:module>
</jaas:config>
```

Passing a data source as a JNDI name

To use an JNDI name, the prefix jndi: needs to be provided. The example below assumes the use of Aries jndi to expose
services via JNDI.

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                 flags="required">
        datasource = jndi:aries:services/
javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
        query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
        query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
    </jaas:module>
</jaas:config>
```

The JDBCLoginModule provides a backend engine allowing:

- add a new user
- delete an user
- list users, roles
- add a new role to an user

- remove a role from an user

NB: the groups are not fully supported by the JDBCBackingEngine.

The following blueprint shows how to define the JDBCLoginModule with the corresponding backend engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0">

    <jaas:config name="karaf">
        <jaas:module
className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                flags="required">
            datasource = jndi:aries:services/
javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
            query.password = SELECT PASSWORD FROM USERS WHERE
USERNAME=?
            query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
            insert.user = INSERT INTO USERS(USERNAME,PASSWORD)
VALUES(?,?)
            insert.role = INSERT INTO ROLES(ROLE,USERNAME)
VALUES(?,?)
            delete.user = DELETE FROM USERS WHERE USERNAME=?
        </jaas:module>
    </jaas:config>

    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory"/>
    </service>

</blueprint>
```

LDAPLoginModule

LoginModule	org.apache.karaf.jaas.modules.ldap.LDAPLoginModule
BackendEngineFactory	N/A

The LDAPLoginModule uses LDAP to load the users and roles and bind the users on the LDAP to check passwords.

The LDAPLoginModule supports the following parameters:

Name	Description

`connection.url`	The LDAP connection URL, e.g. ldap://hostname
`connection.username`	Admin username to connect to the LDAP. This parameter is optiona if it's not provided, the LDAP connection will be anonymous.
`connection.password`	Admin password to connect to the LDAP. Only used if the `connection.username` is specified.
`user.base.dn`	The LDAP base DN used to looking for user, e.g. ou=user,dc=apache,dc=org
`user.filter`	The LDAP filter used to looking for user, e.g. (uid=%u) where %u w be replaced by the username.
`user.search.subtree`	If "true", the user lookup will be recursive (SUBTREE). If "false", the user lookup will be performed only at the first level (ONELEVEL).
`role.base.dn`	The LDAP base DN used to looking for roles, e.g. ou=role,dc=apache,dc=org
`role.filter`	The LDAP filter used to looking for user's role, e.g. (member:=uid=%u)
`role.name.attribute`	The LDAP role attribute containing the role string used by Karaf, e.g cn
`role.search.subtree`	If "true", the role lookup will be recursive (SUBTREE). If "false", the role lookup will be performed only at the first level (ONELEVEL).
`role.mapping`	Define a mapping between roles defined in the LDAP directory for the user, and corresponding roles in Karaf. The format is ldapRole1=karafRole1,karafRole2;ldapRole2=karafRole3,karafRole
`authentication`	Define the authentication backend used on the LDAP server. The default is simple.
`initial.context.factory`	Define the initial context factory used to connect to the LDAP serve The default is com.sun.jndi.ldap.LdapCtxFactory
`ssl`	If "true" or if the protocol on the `connection.url` is `ldaps`, an SS connection will be used
`ssl.provider`	The provider name to use for SSL
`ssl.protocol`	The protocol name to use for SSL (SSL for example)
`ssl.algorithm`	The algorithm to use for the KeyManagerFactory and TrustManagerFactory (PKIX for example)
`ssl.keystore`	The key store name to use for SSL. The key store must be deploye using a `jaas:keystore` configuration.

`ssl.keyalias`	The key alias to use for SSL
`ssl.truststore`	The trust store name to use for SSL. The trust store must be deployed using a `jaas:keystore` configuration.

A example of LDAPLoginModule usage follows:

```
<jaas:config name="karaf">
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
        connection.url = ldap://localhost:389
        user.base.dn = ou=user,dc=apache,dc=org
        user.filter = (cn=%u)
        user.search.subtree = true
        role.base.dn = ou=group,dc=apache,dc=org
        role.filter = (member:=uid=%u)
        role.name.attribute = cn
        role.search.subtree = true
        authentication = simple
  </jaas:module>
</jaas:config>
```

If you wish to use an SSL connection, the following configuration can be used as an example:

```
<ext:property-placeholder />

<jaas:config name="karaf" rank="1">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
        connection.url = ldaps://localhost:10636
        user.base.dn = ou=users,ou=system
        user.filter = (uid=%u)
        user.search.subtree = true
        role.base.dn = ou=groups,ou=system
        role.filter = (uniqueMember=uid=%u)
        role.name.attribute = cn
        role.search.subtree = true
        authentication = simple
        ssl.protocol=SSL
        ssl.truststore=ks
        ssl.algorithm=PKIX
    </jaas:module>
```

```
</jaas:config>

<jaas:keystore name="ks"
               path="file:///${karaf.home}/etc/trusted.ks"
               keystorePassword="secret" />
```

The LDAPLoginModule supports the following patterns that you can use in the filter (user and role filters):

- `%u` is replaced by the user
- `%dn` is replaced by the user DN
- `%fqdn` is replaced by the user full qualified DN (`userDNNamespace`).

For instance, the following configuration will work properly with ActiveDirectory (adding the ActiveDirectory to the
default `karaf` realm):

```
<jaas:config name="karaf" rank="2">
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connection.username=admin
    connection.password=xxxxxxx
    connection.protocol=
    connection.url=ldap://activedirectory_host:389
    user.base.dn=ou=Users,ou=there,DC=local
    user.filter=(sAMAccountName=%u)
    user.search.subtree=true
    role.base.dn=ou=Groups,ou=there,DC=local
    role.name.attribute=cn
    role.filter=(member=%fqdn)
    role.search.subtree=true
    authentication=simple
  </jaas:module>
</jaas:config>
```

NB: the LDAPLoginModule doesn't provide backend engine. It means that the administration of the users and roles should be
performed directly on the LDAP backend.

SyncopeLoginModule

LoginModule	org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule

BackendEngineFactory	org.apache.karaf.jaas.modules.syncope.SyncopeBackendEngineFactory

The Syncope login module uses the Syncope REST API to authenticate users and retrieve the roles.

The Syncope login module just requires one parameter:

Name	Description
address	Location of the Syncope REST API
admin.user	Admin username to administrate Syncope (only required by the backend engine)
admin.password	Admin password to administrate Syncope (only required by the backend engine)

The following snippet shows how to use Syncope with the karaf realm:

```
<jaas:config name="karaf" rank="2">
  <jaas:module
className="org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule"
flags="required">
    address=http://localhost:9080/syncope/cxf
    admin.user=admin
    admin.password=password
  </jaas:module>
</jaas:config>
```

SyncopeLoginModule comes with a backend engine allowing to manipulate users and roles. You have to register the SyncopeBackendEngineFactory service.

For security reason, the SyncopeLoginModule backend engine allows only to list users and roles. You can't create or delete users and roles directly from Karaf. To do it, you have to use the Syncope web console.

For instance, the following blueprint descriptor enables the SyncopeLoginModule and the backend engine factory:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <jaas:config name="karaf" rank="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule"
```

```
                    flags="required">
            address=http://localhost:9080/syncope/cxf
            admin.user=admin
            admin.password=password
        </jaas:module>
    </jaas:config>

    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.syncope.SyncopeBackingEngineFactory"/>
    </service>

</blueprint>
```

ENCRYPTION SERVICE

The EncryptionService is a service registered in the OSGi registry providing means to encrypt and check encrypted passwords. This service acts as a factory for Encryption objects actually performing the encryption.

This service is used in all Karaf login modules to support encrypted passwords.

Configuring properties

Each login module supports the following additional set of properties:

Name	Description
`encryption.name`	Name of the encryption service registered in OSGi (cf. paragraph below)
`encryption.enabled`	Boolean used to turn on encryption
`encryption.prefix`	Prefix for encrypted passwords
`encryption.suffix`	Suffix for encrypted passwords
`encryption.algorithm`	Name of an algorithm to be used for hashing, like "MD5" or "SHA-1"
`encryption.encoding`	Encrypted passwords encoding (can be `hexadecimal` or `base64`)
`role.policy`	A policy for identifying roles (can be `prefix` or `group`) below)
`role.discriminator`	A discriminator value to be used by the role policy

A simple example follows:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = $[karaf.base]/etc/users.properties
        encryption.enabled = true
        encryption.algorithm = MD5
        encryption.encoding = hexadecimal
    </jaas:module>
</jaas:config>
```

Prefix and suffix

The login modules have the ability to support both encrypted and plain passwords at the same time. In some cases, some login modules may be able to encrypt the passwords on the fly and save them back in an encrypted form.

Jasypt

Karaf default installation comes with a simple encryption service which usually fullfill simple needs. However, in some
cases, you may want to install the Jasypt library which provides stronger encryption algorithms
and more control over them.

To install the Jasypt library, the easiest way is to install the available feature:

```
karaf@root> features:install jasypt-encryption
```

It will download and install the required bundles and also register an `EncryptionService` for Jasypt in the OSGi registry.

When configuring a login module to use Jasypt, you need to specify the `encryption.name` property and set it to a value of `jasypt` to make sure the Jasypt encryption service will be used.

In addition to the standard properties above, the Jasypt service provides the following parameters:

Name	Description
`providerName`	Name of the `java.security.Provider` name to use for obtaining the digest algorithm

`providerClassName`	Class name for the security provider to be used for obtaining the digest algorithm
`iterations`	Number of times the hash function will be applied recursively
`saltSizeBytes`	Size of the salt to be used to compute the digest
`saltGeneratorClassName`	Class name of the salt generator

A typical realm definition using Jasypt encryption service would look like:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = $[karaf.base]/etc/users.properties
        encryption.enabled = true
        encryption.name = jasypt
        encryption.algorithm = SHA-256
        encryption.encoding = base64
        encryption.iterations = 100000
        encryption.saltSizeBytes = 16
    </jaas:module>
</jaas:config>
```

Using encrypted property placeholders

When using blueprint framework for OSGi for configuring devices that requires passwords like JDBC datasources,
it is undesirable to use plain text passwords in configuration files. To avoid this problem it is good to store database
passwords in encrypted format and use encrypted property placeholders when ever possible.

Encrypted properties can be stored in plain properties files. The encrypted content is wrapped by an ENC() function.

```
#db.cfg / db.properties
db.url=localhost:9999
db.username=admin
db.password=ENC(zRM7Pb/NiKyCalroBz8CKw==)
```

The encrypted property placeholders can be used either by defining Apache Aries ConfigAdmin `property-placeholder`
or by directly using the Apache Karaf `property-placeholder`. It has one child

element `encryptor` that contains
the actual Jasypt configuration. For detailed information on how to configure the different Jasypt encryptors, see the
Jasypt documentation.

A typical definition using Jasypt encryption would look like:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/
blueprint-cm/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0"
           xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <!-- Configuration via ConfigAdmin property-placeholder -->
  <!-- the etc/*.cfg can contain encrypted values with ENC()
function -->
  <cm:property-placeholder persistent-id="db"
update-strategy="reload">
    <cm:default-properties>
      <cm:property name="encoded" value="ENC(${foo})"/>
    </cm:default-properties>
  </cm:property-placeholder>

  <!-- Configuration via properties file -->
  <!-- Instead of ConfigAdmin, we can load "regular" properties
file from a location -->
  <!-- Again, the db.properties file can contain encrypted values
with ENC() function -->
  <ext:property-placeholder>
    <ext:location>file:etc/db.properties</ext:location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor
class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
      <property name="config">
        <bean
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES"/>
          <property name="passwordEnvName"
value="ENCRYPTION_PASSWORD"/>
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
```

```
  <!-- ... -->

</blueprint>
```

Don't forget to install the jasypt feature to add the support of the enc namespace:

```
karaf@root()> feature:install jasypt-encryption
```

ROLE DISCOVERY POLICIES

The JAAS specification does not provide means to distinguish between User and Role Principals without referring to the
specification classes. In order to provide means to the application developer to decouple the application from Karaf
JAAS implementation role policies have been created.

A role policy is a convention that can be adopted by the application in order to identify Roles, without depending from the implementation. Each role policy can be cofigured by setting a "role.policy" and "role.discriminator" property to the login module configuration. Currently, Karaf provides two policies that can be applied to all Karaf Login Modules.

1. Prefixed Roles
2. Grouped Roles

Prefixed Roles

When the prefixed role policy is used the login module applies a configurable prefix *(property role.discriminator)* to
the role, so that the application can identify the role's principals by its prefix. Example:

```
<jaas:config name="karaf">
    <jaas:module
classNname="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = $[karaf.base]/etc/users.properties
        role.policy = prefix
        role.discriminator = ROLE_
    </jaas:module>
</jaas:config>
```

The application can identify the role principals using a snippet like this:

```
LoginContext ctx = new LoginContext("karaf", handler);
ctx.login();
```

```
authenticated = true;
subject = ctx.getSubject();
for (Principal p : subject.getPrincipals()) {
           if (p.getName().startsWith("ROLE_")) {

roles.add((p.getName().substring("ROLE_".length())));
           }
}
```

Grouped Roles

When the group role policy is used the login module provides all roles as members of a group with a configurable name *(property role.discriminator)*. Example:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = $[karaf.base]/etc/users.properties
        role.policy = group
        role.discriminator = ROLES
    </jaas:module>
</jaas:config>
```

```
LoginContext ctx = new LoginContext("karaf", handler);
ctx.login();
authenticated = true;
subject = ctx.getSubject();
for (Principal p : subject.getPrincipals()) {
    if ((p instanceof Group) &&
("ROLES".equalsIgnoreCase(p.getName()))) {
        Group g = (Group) p;
        Enumeration<? extends Principal> members = g.members();
        while (members.hasMoreElements()) {
            Principal member = members.nextElement();
            roles.add(member.getName());
        }
    }
}
```

DEFAULT ROLE POLICIES

The previous section describes how to leverage role policies. However, Karaf provides a default role policy, based on the following class names:

- org.apache.karaf.jaas.modules.UserPrincipal
- org.apache.karaf.jaas.modules.RolePrincipal
- org.apache.karaf.jaas.modules.GroupPrincipal

It allows you to directly handling the role class:

```
String rolePrincipalClass =
"org.apache.karaf.jaas.modules.RolePrincipal";

for (Principal p : subject.getPrincipals()) {
        if (p.getClass().getName().equals(rolePrincipalClass)) {
                roles.add(p.getName());
        }
}
```

Troubleshooting, Debugging, Profiling, and Monitoring

TROUBLESHOOTING

Logging

Logging is easy to control through the console, with commands grouped under *log* shell. To learn about the available
logging commands type:

```
karaf@root> log<tab>

log:display                log:display-exception
log:get                    log:set
karaf@root>
```

Typical usage is:

Use `log:set` to dynamically change the global log level
Execute the problematic operation
Use `log:display` (or `log:display-exception` to display the log

Worst Case Scenario

If you end up with a Karaf in a really bad state (i.e. you can not boot it anymore) or you just want to revert to a
clean state quickly, you can safely remove the `data` directory just in the installation directory. This folder
contains transient data and will be recreated if removed when you relaunch Karaf.
You may also want to remove the files in the `deploy` folder to avoid them being automatically installed when Karaf
is started the first time.

DEBUGGING

Usually, the easiest way to debug Karaf or any application deployed onto it is to use remote debugging.

Remote debugging can be easily activated by using the `debug` parameter on the command line.

```
> bin/karaf debug
```

or on Windows

```
> bin\karaf.bat debug
```

Another option is to set the `KARAF_DEBUG` environment variable to `TRUE`.

This can be done using the following command on Unix systems:

```
export KARAF_DEBUG=true
```

On Windows, use the following command

```
set KARAF_DEBUG=true
```

Then, you can launch Karaf using the usual way:

```
bin/karaf
```

or

```
bin\karaf.bat
```

Last, inside your IDE, connect to the remote application (the default port to connect to is 5005).

This option works fine when it is needed to debug a project deployed top of Apache Karaf. Nervertheless, you will be blocked
if you would like to debug the server Karaf. In this case, you can change the following parameter suspend=y in the
karaf.bat script file. That will cause the JVM to pause just before running main() until you attach a debugger then it
will resume the execution. This way you can set your breakpoints anywhere in the code and you should hit them no matter
how early in the startup they are.

```
export DEFAULT_JAVA_DEBUG_OPTS='-Xdebug -Xnoagent
-Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005'
```

and on Windows,

```
set DEFAULT_JAVA_DEBUG_OPTS='-Xdebug -Xnoagent
-Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005'
```

Debugging Environment Variables

Variable	Description	Default
KARAF_DEBUG	Set to TRUE to enable debugging.	
EXTRA_JAVA_OPTS	Java options append to JAVA_OPTS	
JAVA_DEBUG_OPTS	Java options to enable debuging.	Calculated based on the OS
JAVA_DEBUG_PORT	Port used by the debugger.	5005

PROFILING

jVisualVM

You have to edit the `etc/config.properties` configuration file to add the jVisualVM package:

```
org.osgi.framework.bootdelegation=...,org.netbeans.lib.profiler.server
```

Run Karaf from the console, and you should now be able to connect using jVisualVM.

YourKit

You need a few steps to be able to profile Karaf using YourKit.

The first one is to edit the `etc/config.properties` configuration file and add the following property:

```
org.osgi.framework.bootdelegation=...,com.yourkit.*
```

Then, set the `JAVA_OPTS` environment variable:

```
export JAVA_OPTS='-Xmx512M -agentlib:yjpagent'
```

or, on Windows

```
set JAVA_OPTS='-Xmx512M -agentlib:yjpagent'
```

Run Karaf from the console, and you should now be able to connect using YourKit standalone or from your favorite IDE.

MONITORING

Karaf uses JMX for monitoring and management of all Karaf components.

The JMX connection could be:

- local using the process id
- remote using the `rmiRegistryPort` property defined in `etc/org.apache.karaf.management.cfg` file.

Using JMX, you can have a clean overview of the running Karaf instance:

- A overview with graphics displaying the load in terms of thread, heap/GC, etc:
- A thread overview:
- A memory heap consumption, including "Perform GC" button:
- A complete JVM summary, with all number of threads, etc:

You can manage Karaf features like you are in the shell. For example, you have access to the Admin service MBean,
allowing you to create, rename, destroy, change SSH port, etc. Karaf instances:

You can also manage Karaf features MBean to list, install, and uninstall Karaf features:

Writing integration tests

We recommend using PAX Exam to write integration tests when developing applications using Karaf.

Starting with Karaf 3.0 we've also included a component briding between Karaf and Pax Exam making it easier to write integration tests for Karaf or Karaf based Distributions such as Servicemix or Geronimo.

INTRODUCTION

To make use of this new framework simply add the following dependencies into your integration tests pom.xml:

```
<!-- Karaf Test Framework Version -->
<dependency>
  <groupId>org.apache.karaf.tooling.exam</groupId>
  <artifactId>org.apache.karaf.tooling.exam.container</artifactId>
  <version>4.0.3</version>
  <scope>test</scope>
</dependency>
<!-- Pax Exam version you would like to use. At least 2.2.x is
required. -->
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-junit4</artifactId>
  <version>4.6.0</version>
  <scope>test</scope>
</dependency>
```

As a next step you need to reference the distribution you want to run your tests on. For example, if you want to run your tests on Karaf the following section would be required in the integration tests pom.xml:

```
<dependency>
  <groupId>org.apache.karaf</groupId>
  <artifactId>apache-karaf</artifactId>
  <version>4.0.3</version>
  <type>zip</type>
```

```
  <scope>test</scope>
</dependency>
```

If you want to make use of Exams "versionAsInProject" feature you also need to add the following section:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.servicemix.tooling</groupId>
      <artifactId>depends-maven-plugin</artifactId>
      <version>1.2</version>
      <executions>
        <execution>
          <id>generate-depends-file</id>
          <goals>
            <goal>generate-depends-file</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

With this done we can start writing our first test case:

```
import static junit.framework.Assert.assertTrue;
import static
org.apache.karaf.tooling.exam.options.KarafDistributionOption.karafDistributionCo
import static org.ops4j.pax.exam.CoreOptions.maven;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.ops4j.pax.exam.Option;
import org.ops4j.pax.exam.junit.Configuration;
import org.ops4j.pax.exam.junit.ExamReactorStrategy;
import org.ops4j.pax.exam.junit.JUnit4TestRunner;
import
org.ops4j.pax.exam.spi.reactors.AllConfinedStagedReactorFactory;

@RunWith(JUnit4TestRunner.class)
@ExamReactorStrategy(AllConfinedStagedReactorFactory.class)
public class VersionAsInProjectKarafTest {
```

```
    @Configuration
    public Option[] config() {
        return new Option[]{
karafDistributionConfiguration().frameworkUrl(

maven().groupId("org.apache.karaf").artifactId("apache-karaf").type("zip").versio
            .karafVersion("2.2.4").name("Apache Karaf")};
    }

    @Test
    public void test() throws Exception {
        assertTrue(true);
    }
}
```

COMMANDS

Basically the Pax Exam - Karaf bridge introduced with 3.0 should support all commands you know from Pax Exam 2.x. In addition we've added various additional commands to make your life easier. Those commands are listed and explained in this sub section.

As a small remark: All of the Options explained here are also accessible via the static methods in the KarafDistributionOption class in the options package automatically on your classpath when you reference the container package.

KarafDistributionConfigurationOption

The framework itself is non of the typical runtimes you define normally in PAXEXAM. Instead you define a packed distribution as zip or tar.gz. Those distributions have to follow the Karaf packaging style. Therefore instead of Karaf you can also enter Servicemix or Geronimo.

```
new KarafDistributionConfigurationOption(
  "mvn:org.apache.karaf/apache-karaf/2.2.4/zip", // artifact to
unpack and use
  "karaf", // name; display only
  "2.2.4") // the karaf version; this one is relevant since the
startup script differs between versions
```

or for Servicemix e.g.

```
new KarafDistributionConfigurationOption(
  "mvn:org.apache.servicemix/apache-servicemix/4.4.0/zip", //
artifact to unpack and use
  "servicemix", // name; display only
  "2.2.4") // the karaf version; this one is relevant since the
startup script differs between versions
```

As an alternative you can also use the maven url resolvers. Please keep in mind that this only works starting with karaf-3.0.0 since there will be problems with the pax-url version. In addition, if you want to make use of the versionAsInProject part you also need to define the following maven-plugin in the pom file of your integration tests:

```
...
<dependency>
  <groupId>org.apache.karaf</groupId>
  <artifactId>apache-karaf</artifactId>
  <type>zip</type>
  <classifier>bin</classifier>
  <scope>test</scope>
</dependency>
...
<plugin>
  <groupId>org.apache.servicemix.tooling</groupId>
  <artifactId>depends-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>generate-depends-file</id>
      <goals>
        <goal>generate-depends-file</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
@Configuration
    public Option[] config() {
        return new Option[]{
karafDistributionConfiguration().frameworkUrl(

maven().groupId("org.apache.karaf").artifactId("apache-karaf").type("zip")
                .classifier("bin").versionAsInProject()) };
    }
```

In addition to the framework specification options this option also includes various additional configuration options. Those options are used to configure the internal properties of the runtime environment.

Unpack Directory

Paxexam-Karaf Testframework extracts the distribution you specify by default into the paxexam config directory. If you would like to unpack them into your target directory simply extend the KarafDistributionConfigurationOption with the unpackDirectoryFile like shown in the next example:

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip")
        .unpackDirectory(new File("target/paxexam/unpack/")) };
}
```

Use Deploy Folder

Karaf distributions come by default with a deploy folder where you can simply drop artifacts to be deployed. In some distributions this folder might have been removed. To still be able to deploy your additional artifacts using default Pax Exam ProvisionOptions you can configure PaxExam Karaf to use a features.xml (which is directly added to your etc/org.apache.karaf.features.cfg) for those deploys. To use it instead of the deploy folder simply do the following:

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip")
        .useDeployFolder(false)) };
}
```

KarafDistributionKitConfigurationOption

The KarafDistributionKitConfigurationOption is almost equal to all variations of the KarafDistributionConfigurationOption with the exception that it requires to have set a platform and optionally the executable and the files which should be made executable

additionally. By default it is bin/karaf for nix platforms and bin\karaf.bat for windows platforms. The executable option comes in handy if you like to e.g. embed an own java runtime. You should add a windows AND a linux Kit definition. The framework automatically takes the correct one then. The following shows a simple example for karaf:

```
@Configuration
public Option[] config() {
    return new Option[]{
        new
KarafDistributionKitConfigurationOption("mvn:org.apache.karaf/
apache-karaf/4.0.3/zip",

Platform.WINDOWS).executable("bin\\karaf.bat").filesToMakeExecutable("bin\\admin.
        new
KarafDistributionKitConfigurationOption("mvn:org.apache.karaf/
apache-karaf/4.0.3/tar.gz", "karaf",
            Platform.NIX).executable("bin/
karaf").filesToMakeExecutable("bin/admin") };
}
```

KarafDistributionConfigurationFilePutOption

The option replaces or adds an option to one of Karaf's configuration files:

```
new KarafDistributionConfigurationFilePutOption(
  "etc/config.properties", // config file to modify based on
karaf.base
  "karaf.framework", // key to add or change
  "equinox") // value to add or change
```

This option could also be used in "batch-mode" via a property file. Therefore use the KarafDistributionOption#editConfigurationFilePut(final String configurationFilePath, File source, String... keysToUseFromSource) method. This option allows you to add all properties found in the file as KarafDistributionConfigurationFilePutOption. If you configure the "keysToUseFromSource" array only the keys specified there will be used. That way you can easily put an entire range of properties.

KarafDistributionConfigurationFileExtendOption

This one does the same as the KarafDistributionConfigurationFilePutOption option with the one difference that it either adds or appends a specific property. This is especially useful if you do not want to store the entire configuration in the line in your code.

This option could also be extended in "batch-mode" via a property file. Therefore use the KarafDistributionOption#editConfigurationFileExtend(final String configurationFilePath, File source, String... keysToUseFromSource) method. This option allows you to extend all properties found in the file as KarafDistributionConfigurationFileExtendOption. If you configure the "keysToUseFromSource" array only the keys specified there will be used. That way you can easily extend an entire range of properties.

KarafDistributionConfigurationFileReplacementOption

The file replacement option allows you to simply replace a file in you Karaf distribution with a different file:

```
new KarafDistributionConfigurationFileReplacementOption("etc/
tests.cfg", new File(
    "src/test/resources/
BaseKarafDefaultFrameworkDuplicatedPropertyEntryTestSecondKey"));
```

ProvisionOption

The new test container fully supports the provision option. Feel free to use any option provided here by paxexam itself (e.g. Maven resolver). All those artifacts are copied into the deploy folder of your Karaf distribution before it is started. Therefore they all will be available after startup.

KarafDistributionConfigurationConsoleOption

The test container supports options to configure if the localConsole and/or the remote shell should be started. Possible options to do so are shown in the following two examples:

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
        configureConsole().ignoreLocalConsole().startRemoteShell()
};
}
```

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
        configureConsole().startLocalConsole(),
configureConsole().ignoreRemoteShell() };
}
```

VMOption

The Karaf container passes the vmOptions now through to the Karaf environment. They are directly passed to the startup of the container. In addition the KarafDistributionOption helper has two methods (debugConfiguration() and debugConfiguration(String port, boolean hold)) to activate debugging quickly.

LogLevelOption

The Paxexam-Karaf specific log-level option allows an easy way to set a specific log-level for the Karaf based distribution. For example simply add the following to your Option[] array to get TRACE logging:

```
import static
org.openengsb.labs.paxexam.karaf.options.KarafDistributionOption.logLevel;
...
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
        logLevel(LogLevel.TRACE) };
}
```

DoNotModifyLogOption

The option to modify the logging behavior requires that the container automatically modifies the logging configuration file. If you would like to suppress this behavior simply set the doNotModifyLogConfiguration option as shown in the next example:

```
@Configuration
public Option[] config() {
```

```
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
        doNotModifyLogConfiguration() };
}
```

KeepRuntimeFolderOption

Per default the test container removes all test runner folders. If you want to keep them for any reasons (e.g. check why a test fails) set the following option:

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
        keepRuntimeFolder() };
}
```

FeaturesScannerProvisionOption

The FeaturesScannerProvisionOption (e.g. CoreOption.scanFeature()) are directly supported by the Paxexam Karaf Testframework.

BootDelegationOption

The BootDelegationOption as known from PaxExam is also supported added the boot delegation string directly into the correct property files.

SystemPackageOption

The Standard Exam SystemPackageOption is implemented by adding those packages to "org.osgi.framework.system.packages.extra" of the config.properties file.

BootClasspathLibraryOption

The BootClasspathLibraryOption is honored by copying the urls into the lib directory where they are automatically taken and worked on.

ExamBundlesStartLevel

The ExamBundlesStartLevel can be used to configure the start lvl of the bundles provided by the test-frameworks features.xml. Simply use it as a new option like:

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
4.0.3/zip"),
            useOwnExamBundlesStartLevel(4) };
}
```

DRIVER

Drivers are the parts of the framework responsible for running the Karaf Based Distribution. By default the already in the overview explained KarafDistributionConfigurationOption uses a JavaRunner starting the distribution platform independent but not using the scripts in the distribution. If you like to test those scripts too an option is to to use the ScriptRunner via the KarafDistributionKitConfigurationOption instead.

JavaRunner

The JavaRunner builds the entire command itself and executes Karaf in a new JVM. This behavior is more or less exactly what the default runner does. Simply use the KarafDistributionConfigurationOption as explained in the Commands section to use this.

ScriptRunner

The script runner has the disadvantage over the java runner that it is also platform dependent. The advantage though is that you can also test your specific scripts. To use it follow the explanation of the KarafDistributionKitConfigurationOption in the Commands section.

Github Contributions

Some people prefer to make contributions to karaf source via github. If you are one of them, this is for you!

INTRODUCTION

Apache Karaf is available as a periodically replicated mirror on: https://github.com/apache/karaf

SUGGESTED WORKFLOW

1. make a fork of karaf repo github mirror
2. do all your new work on your own karaf fork
3. when ready, file a jira issue https://issues.apache.org/jira/browse/KARAF, attach the link to your github pull request, and ask for a review
4. one of karaf committers will discuss your pull request on github; and at some point your pull request will be accepted
5. when your pull request is accepted, squash it into a single commit and attach single patch file to the original jira, with ASF grant check box selected
6. now pray to your favorite ASF committer to really accept the patch :-)
7. when your patch is committed to the svn, and you can verify it in the latest karaf snapshot, close your pull request on github

LICENSE REMINDER

in order for your contributions to be accepted:

- all files must contain ASL license grant header
- you must select ASF grant check box when attaching patch to the jira

HOW TO GENERATE A ONE-FILE-PATCH VIA THROW-AWAY BRANCH

here is one way to generate squash of your commits:

http://stackoverflow.com/questions/616556/how-do-you-squash-commits-into-one-patch-with-git-format-patch

```
#
# 'archon' referers to karaf mirror
# 'origin' referers to your own fork
#

# attach karaf mirror as remote, if not done yet
git remote add archon https://github.com/apache/karaf

# fetch latest karaf mirror
git fetch archon

# ensure you are on your fork trunk
git checkout origin/trunk

# kill previous patch delivery, if you had one
git branch -D delivery

# make new delivery throw-away branch, based on latest karaf
mirror
git branch delivery archon/trunk

# use it
git checkout delivery

# squash all your local development into a single commit
git merge --squash trunk

# commit it to the delivery branch
git commit -m "delivery"

# generate a patch file against the mirror
git format-patch archon/trunk
```

root of your karaf source now contains a file named "0001-delivery.patch.txt" (please attach the .txt ending;this will allow commiters to open your patch directly in the browser and give it a short look there) which you should attach to your karaf jira, and ask to commit to the svn trunk