Apache Karaf
Version 2.3.12

Apache Karaf
Users' Guide

Table of contents

Overview
Quick Start
Users Guide
Developers Guide

Overview

Karaf Overview

Apache Karaf is a small OSGi based runtime which provides a lightweight container onto which various components and applications can be deployed.

Here is a short list of features supported by the Karaf:

- **Hot deployment**: Karaf supports hot deployment of OSGi bundles by monitoring jar files inside the `deploy` directory. Each time a jar is copied in this folder, it will be installed inside the runtime. You can then update or delete it and changes will be handled automatically. In addition, Karaf also supports exploded bundles and custom deployers (Blueprint and Spring ones are included by default).
- **Dynamic configuration**: Services are usually configured through the ConfigurationAdmin OSGi service. Such configuration can be defined in Karaf using property files inside the `etc` directory. These configurations are monitored and changes on the properties files will be propagated to the services.
- **Logging System**: using a centralized logging back end supported by Log4J, Karaf supports a number of different APIs (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, OSGi)
- **Provisioning**: Provisioning of libraries or applications can be done through a number of different ways, by which they will be downloaded locally, installed and started.
- **Native OS integration**: Karaf can be integrated into your own Operating System as a service so that the lifecycle will be bound to your Operating System.
- **Extensible Shell console**: Karaf features a nice text console where you can manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.
- **Remote access**: use any SSH client to connect to Karaf and issue commands in the console
- **Security framework** based on JAAS
- **Managing instances**: Karaf provides simple commands for managing multiple instances. You can easily create, delete, start and stop instances of Karaf through the console.
- Supports the latest OSGi 4.2 containers: Apache Felix Framework 3.0 and Eclipse Equinox 3.6

Quick Start

1. Quick Start

If you are in a hurry to have Apache Karaf up and running right away, this section will provide you with some basic steps for downloading, building (when needed) and running the server in no time. This is clearly not a complete guide so you may want to check other sections of this guide for further information.

All you need is 5 to 10 minutes and to follow these basic steps.
- Background
- Getting the software
- Start the server
- Deploy a sample application

BACKGROUND

Apache Karaf is a small and lightweight OSGi based runtime. This provides a small lightweight container onto which various bundles can be deployed.

Apache Karaf started life as the Apache ServiceMix kernel and then moved as a Apache Felix subproject before becoming a top level project.

GETTING THE SOFTWARE

At this time you have one option to get the software. The fastest and easiest way is to get the binary directly from the Apache site. Since this article is intended to help you to have Apache Karaf up and running in the fastest way only the binary download will be covered at this time.

Prerequisites

Although this installation path is the fastest one, you will still need to install some software before installing Karaf.

Karaf requires a Java SE 5 environment to run. Refer to http://www.oracle.com/technetwork/java/javase/ for details on how to download and install Java SE 1.5 or greater.

Download binaries

You will need to select the appropriate installation image for the platform you're using to run Karaf. Open a Web browser and access the following URL, there you will find the available packages for download (binaries and source code).

http://karaf.apache.org/index/community/download.html

Select the file compression format compatible with your system (zip for windows, tar.gz for unixes) by clicking directly on the link, download it and expand the binary to your hard drive in a new directory; for example in z:\karaf - from now on this directory will be referenced as <KARAF_HOME>. Please remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.

The installation of Karaf is as simple as uncompressing the .zip or .tar.gz files. The next step is to start the server.

START THE SERVER

With Karaf already installed, open a command line console and change 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:

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

```
karaf@root>
```

```
admin:change-port        admin:connect
admin:create             admin:destroy
admin:list               admin:start
admin:stop               config:cancel
config:edit              config:list
config:propappend        config:propdel
config:proplist          config:propset
config:update            dev:dynamic-import
dev:framework            dev:print-stack-traces
dev:show-tree            features:addUrl
features:info            features:install
features:list            features:listUrl
features:refreshUrl      features:removeUrl
```

```
features:uninstall        log:display
log:display-exception     log:get
log:set                   osgi:bundle-level
osgi:headers              osgi:install
osgi:list                 osgi:ls
osgi:refresh              osgi:resolve
osgi:restart              osgi:shutdown
osgi:start                osgi:start-level
osgi:stop                 osgi:uninstall
osgi:update               packages:exports
packages:imports          shell:cat
shell:clear               shell:each
shell:echo                shell:exec
shell:grep                shell:history
shell:if                  shell:info
shell:java                shell:logout
shell:new                 shell:printf
shell:sleep               shell:sort
shell:tac                 ssh:ssh
ssh:sshd                  cat
clear                     each
echo                      exec
grep                      history
if                        info
java                      logout
new                       printf
sleep                     sort
tac                       bundle-level
headers                   install
list                      ls
refresh                   resolve
restart                   shutdown
start                     start-level
stop                      uninstall                update
karaf@root>
```

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

```
karaf@root> admin:create --help
DESCRIPTION
        admin:create

        Create a new instance.
```

```
SYNTAX
        admin:create [options] name

ARGUMENTS
        name
                The name of the new container instance

OPTIONS
        --help
                Display this help message
        -f, --feature
                Initial features. This option can be specified
multiple times to enable multiple initial
                features
        -p, --port
                Port number for remote shell connection
        -l, --location
                Location of the new container instance in the
file system
        -furl, --featureURL
                Additional feature descriptor URLs. This option
can be specified multiple times to add
                multiple URLs

karaf@root>
```

Note that the console supports tab completion, so you just need to enter `ad <tab> cr <tab>` instead of `admin:create`.

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:

```
features:addurl mvn:org.apache.camel/camel-example-osgi/2.10.0/
xml/features
features:install camel-example-osgi
```

The example installed is using Camel to start a timer every 2 seconds and output a message on the console.

The previous commands download the Camel features descriptor and install the example feature.

```
>>>> SpringDSL set body:  Fri Jan 07 11:59:51 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:53 CET 2011
>>>> SpringDSL set body:  Fri Jan 07 11:59:55 CET 2011
```

Stopping and uninstalling the sample application

To stop this demo, run the following command:

```
features:uninstall camel-example-osgi
```

Common Problems

1. Launching Karaf can result in a deadlock in Felix during module dependency resolution.
   This is often a result of sending a SIGINT (control-C) to the process when it will not cleanly exit.
   This can corrupt the caches and cause startup problems in the very next launch. It is fixed by emptying the component cache:

   ```
   rm -rf data/cache/*
   ```

STOPPING KARAF

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

```
^D
```

Alternatively, you can also run the following command:

```
system:shutdown
```

SUMMARY

This document showed how simple it is to have Apache Karaf up and running. The overall time for getting the server running should be less than five
minutes if you have the prerequisite (Java 1.5) already installed. Additionally, this article also showed you how to deploy and test a simple
Apache Camel application.

Users Guide

Installation

This chapter describes how to install Apache Karaf for both Unix and Windows platforms, including
prerequisite software and necessary download links.

PRE-INSTALLATION REQUIREMENTS

Hardware:
- 20 MB of free disk space for the Apache Karaf x.y binary distribution.

Operating Systems:
- Windows: Windows Vista, Windows XP SP2, Windows 2000.
- Unix: Ubuntu Linux, Powerdog Linux, MacOS, AIX, HP-UX, Solaris, any Unix platform that supports Java.

Environment:
- Java SE Development Kit 1.5.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, e.g., `c:\Program Files\jdk.1.5.0_06`. To accomplish that, press Windows key and Break key together, switch to "Advanced" tab and click on "Environment Variables". Here, check for the variable and, if necessary, add it.

BUILDING FROM SOURCES

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

Hardware:
- 200 MB of free disk space for the Apache Karaf x.y source distributions or git clone, the Maven build and the dependencies Maven downloads.

Environment:
- Java SE Developement Kit 1.5.x or greater (http://www.oracle.com/technetwork/java/javase/).
- Apache Maven 2.2.1 (http://maven.apache.org/download.html).

Building on Windows

This procedure explains how to download and install the source distribution on a Windows system. **NOTE:** Karaf requires Java 5 is compile, build and run.

1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Scroll down to the "Apache Karaf" section and select the desired distribution. For a source distribution, the filename will be similar to: `apache-karaf-x.y-src.zip`.
3. Extract Karaf from the ZIP file into a directory of your choice. Please remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.
4. Build Karaf using Maven 2.2.1 or greater and Java 5.
   The recommended method of building Karaf is the following:

   ```
   cd [karaf_install_dir]\src
   ```

   where `karaf_install_dir` is the directory in which Karaf was installed.

   ```
   mvn
   ```

   Both steps take around 10 to 15 minutes.
5. Unzip the distribution using your favorite zip tool. The windows distribution is available at

   ```
   [karaf_install_dir]\assembly\target\apache-karaf-x.y.zip
   ```

6. Proceed to the Starting Karaf chapter.

Building on Unix

This procedure explains how to download and install the source distribution on a Unix system. This procedure assumes the Unix machine has a browser. Please see the previous Unix Binary Installation section for ideas on how to install Karaf without a browser. **NOTE:** Karaf requires Java 5 to compile, build and run.

1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Scroll down to the "Apache Karaf" section and select the desired distribution. For a source distribution, the filename will be similar to: `apache-karaf-x.y-src.tar.gz`.
3. Extract the files from the ZIP file into a directory of your choice. For example:

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

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

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

where `karaf_install_dir` is the directory in which Karaf was installed.

```
mvn
```

5. Uncompress the distribution that has just been created

```
cd [karaf_install_dir]/assembly/target
gunzip apache-karaf-x.y.tar.gz
tar xvf apache-karaf-x.y.tar
```

6. Proceed to the Starting Karaf chapter.

INSTALLATION PROCEDURE FOR WINDOWS

This procedure explains how to download and install the binary distribution on a Windows system.
1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Scroll down to the "Apache Karaf" section and select the desired distribution. For a binary distribution, the filename will be similar to: `apache-karaf-x.y.zip`.
3. Extract the files from the ZIP file into a directory of your choice. Please remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.
4. Proceed to the Starting Karaf chapter.
5. Optional: see enabling Colorized Console Output On Windows

> **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 PROCEDURE FOR UNIX

This procedure explains how to download and install the binary distribution on a Unix system.

1. From a browser, navigate to http://karaf.apache.org/index/community/download.html.
2. Scroll down to the "Apache Karaf" section and select the desired distribution. For a binary Unix distribution, the filename will be similar to: apache-karaf-x.y.tar.gz.
3. Extract the files from the gzip file into a directory of your choice. For example:

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

   Please remember the restrictions concerning illegal characters in Java paths, e.g. !, % etc.
4. Proceed to the Starting Karaf chapter.

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 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 5 or 6 installation.

Directory structure

The directory layout of a Karaf installation is as follows:
- `/bin`: startup scripts
- `/etc`: configuration files
- `/data`: working directory
    - `/cache`: OSGi framework bundle cache
    - `/generated-bundles`: temporary folder used by the deployer
    - `/log`: log files
- `/deploy`: hot deploy directory
- `/instances`: directory containing child instances
- `/lib`: contains the bootstrap libraries
    - `/lib/ext`: directory for JRE extensions
    - `/lib/endorsed`: directory for endorsed libraries
- `/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.

Starting and Stopping Karaf

This chapter describes how to start and stop Apache Karaf and the various options that are available.

STARTING KARAF

On Windows

From a console window, change to the installation directory and run `Karaf`. For the binary distribution, go to

```
cd [karaf_install_dir]
```

where `karaf_install_dir` is the directory in which Karaf was installed, e.g., `c:\Program Files\apache-karaf-x.y`.

Then type:

```
bin\karaf.bat
```

On Unix

From a command shell, change to the installation directory and run `Karaf`. For the binary distribution, go to

```
cd [karaf_install_dir]
```

where `karaf_install_dir` is the directory in which Karaf was installed, e.g., `/usr/local/apache-karaf-x.y`.

Then type:

```
bin/karaf
```

> **Warning**
> Do NOT close the console or shell in which Karaf was started, as that will terminate Karaf (unless Karaf was started with nohup).

STARTING KARAF WITHOUT CONSOLE

Karaf can be started without the console if you don't intend to use it (one can always connect using the remote ssh access) using the following command:

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

or, on Unix:

```
bin\karaf server
```

STARTING KARAF IN THE BACKGROUND

Karaf can be easily started as a background process using the following command:

```
bin\start.bat
```

or, on Unix:

```
bin\start
```

STARTING KARAF FROM CLEAN

Karaf can be reset to a clean state by simply deleting the `data` folder.
For convenience, a parameter on the `karaf` and `start` scripts is available:

```
bin/start clean
```

STOPPING KARAF

For both Windows and Unix installations, you can perform a clean shutdown of Karaf by using the following command when inside a Karaf console:

```
system:shutdown
```

or simply:

```
shutdown
```

The shutdown command asks you to confirm that you really want to shutdown. If you are sure about the shutdown and avoid the confirmation message, you can use the -f or --force option:

```
system:shutdown -f
```

It's also possible to delay the shutdown using the time argument. 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, in which m is the number of minutes to wait. The work now is an alias for +0.

The following command will shutdown Karaf at 10:35am:

```
system:shutdown 10:35
```

The following command will shutdown Karaf in 10 minutes:

```
system:shutdown +10
```

If you're running from the main console, exiting the shell using `logout` or `Ctrl+D` will also terminate the Karaf instance.

From a command shell, you can run the following command:

```
bin\stop.bat
```

or, on Unix:

```
bin/stop
```

Service Wrapper

INTRODUCTION

The Karaf Wrapper (for service wrapper) makes it possible to install Karaf as a Windows Service. Likewise, the scripts shipped with Karaf also make it very easy to install Karaf as a daemon process on Unix systems.

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

SUPPORTED PLATFORMS

The following platforms are supported by the Karaf Wrapper:
- AIX
- FreeBSD
- HP-UX, 32-bit and 64-bit versions
- SGI Irix
- Linux kernels 2.2.x, 2.4.x, 2.6.x. Known to work with Debian, Ubuntu, and Red Hat, but should work with any distribution. Currently supported on both 32-bit and 64-bit x86, Itanium, and PPC systems.
- Macintosh OS X
- Sun OS, Solaris 9 and 10. Currently supported on both 32-bit and 64-bit sparc, and x86 systems.
- Windows - Windows 2000, XP, 2003, Vista, 2008 and Windows 7. Currently supported on both 32-bit and 64-bit x86 and Itanium systems. Also known to run on Windows 98 and ME, however due the lack of support for services in the OS, the Wrapper can be run only in console mode.

INSTALLATION

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

```
karaf@root> features: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 )
```

Using `wrapper:install`, you can install Karaf as a service.

For instance, to register Karaf as a service (depending of the running OS), in automatic start mode, simply type:

```
karaf@root> wrapper:install -s AUTO_START -n KARAF -d Karaf -D
"Karaf Service"
```

For instance, on Linux, `wrapper:install` command will do:

```
karaf@root> wrapper:install -s AUTO_START -n KARAF -d Karaf -D
"Karaf Service"
Creating file: /home/onofreje/apache-karaf-2.1.3/bin/
KARAF-wrapper
Creating file: /home/onofreje/apache-karaf-2.1.3/bin/
KARAF-service
Creating file: /home/onofreje/apache-karaf-2.1.3/etc/
KARAF-wrapper.conf
Creating file: /home/onofreje/apache-karaf-2.1.3/lib/
libwrapper.so
Creating file: /home/onofreje/apache-karaf-2.1.3/lib/
```

```
karaf-wrapper.jar
Creating file: /home/onofreje/apache-karaf-2.1.3/lib/
karaf-wrapper-main.jar

Setup complete.  You may wish to tweak the JVM properties in the
wrapper configuration file:
        /home/onofreje/apache-karaf-2.1.3/etc/KARAF-wrapper.conf
before installing and starting the service.

The way the service is installed depends upon your flavor of
Linux.

On Redhat/Fedora/CentOS Systems:
  To install the service:
    $ ln -s /home/onofreje/apache-karaf-2.1.3/bin/KARAF-service
/etc/init.d/
    $ chkconfig KARAF-service --add

  To start the service when the machine is rebooted:
    $ chkconfig KARAF-service on

  To disable starting the service when the machine is rebooted:
    $ chkconfig KARAF-service off

  To start the service:
    $ service KARAF-service start

  To stop the service:
    $ service KARAF-service stop

  To uninstall the service :
    $ chkconfig KARAF-service --del
    $ rm /etc/init.d/KARAF-service

On Ubuntu/Debian Systems:
  To install the service:
    $ ln -s /home/onofreje/apache-karaf-2.1.3/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
```

CONFIGURATION HINTS

By default, the wrapper sets the JAVA_HOME in etc/karaf-wrapper.conf file. However, the bin/karaf-service script uses java. If required, you have to define the JAVA_HOME in this bin/karaf-service script.

   If you need to append parameters to the "java" invoke (like memory configurations) you add those in the KARAF-wrapper file using "wrapper.java.additional.n=PARAMETER" where "n" is the number of the additional config (typically you simply look for the last entry and use n+1) and PARAMETER is any JVM parameter you would like to append, such as "-XX:MaxPermSize=1024m".

Using the console

VIEWING AVAILABLE COMMANDS

To see a list of the available commands in the console press the `<tab>` key at the prompt.

```
root@root> Display all 182 possibilities? (y or n)
*:help
addurl                          admin:change-opts
admin:change-rmi-registry-port
admin:change-ssh-port           admin:connect
admin:create
admin:destroy                   admin:list
admin:rename
admin:start                     admin:stop
bundle-level
cancel                          cat
change-opts
change-rmi-registry-port        change-ssh-port
clear
commandlist                     config:cancel
config:edit
config:list                     config:propappend
config:propdel
config:proplist                 config:propset
config:update
connect                         create
create-dump
destroy                         dev:create-dump
dev:dynamic-import
dev:framework                   dev:print-stack-traces
dev:restart
dev:show-tree                   dev:watch
display
display-exception               dynamic-import
each
echo                            edit
exec
exports                         features:addurl
```

```
features:info
features:install              features:list
features:listrepositories
features:listurl              features:listversions
features:refreshurl
features:removerepository     features:removeurl
features:uninstall
framework                     get
grep
head                          headers
help
history                       if
imports
info                          install
jaas:cancel
jaas:commandlist              jaas:list
jaas:manage
jaas:roleadd                  jaas:roledel
jaas:update
jaas:useradd                  jaas:userdel
jaas:userlist
java                          list
listrepositories
listurl                       listversions
log:clear
log:display                   log:display-exception
log:get
log:set                       log:tail
logout
ls                            manage
more
new                           osgi:bundle-level
osgi:headers
osgi:info                     osgi:install
osgi:list
osgi:ls                       osgi:refresh
osgi:resolve
osgi:restart                  osgi:shutdown
osgi:start
osgi:start-level              osgi:stop
osgi:uninstall
osgi:update                   packages:exports
packages:imports
```

```
print-stack-traces              printf
propappend
propdel                         proplist
propset
refresh                         refreshurl
removerepository
removeurl                       rename
resolve
restart                         roleadd
roledel
set                             shell:cat
shell:clear
shell:each                      shell:echo
shell:exec
shell:grep                      shell:head
shell:history
shell:if                        shell:info
shell:java
shell:logout                    shell:more
shell:new
shell:printf                    shell:sleep
shell:sort
shell:tac                       shell:tail
show-tree
shutdown                        sleep
sort
ssh                             ssh:ssh
ssh:sshd
sshd                            start
start-level
stop                            tac
tail
uninstall                       update
useradd
userdel                         userlist
watch
root@root>
```

The `<tab>` key toggles autocompletion anywhere on the line, so if you want to see the commands in the `osgi` group, type the first letters and hit `<tab>`. Depending on the commands, autocompletion may be available for options and arguments too.

GETTING HELP FOR A COMMAND

To view help on a particular command, type the command followed by `--help` or use the `help` command followed by the name of the command:

```
karaf@root> features:list --help
DESCRIPTION
        features:list

        Lists all existing features available from the defined
repositories.

SYNTAX
        features:list [options]

OPTIONS
        --help
                        Display this help message
        -i, --installed
                        Display a list of all installed features
only
```

MORE...

The list of all available commands and their usage is also available in a dedicated section.

You'll find a more in-depth guide to the shell syntax in the developers guide.

The console can also be easily extended by creating new commands as explained in the developers guide.

Using remote instances

CONFIGURING REMOTE INSTANCES

It does not always make sense to manage an instance of Karaf using its local console. You can manage Karaf remotely using a remote console.

When you start Karaf, it enables a remote console that can be accessed over SSH from any other Karaf console or plain SSH client. The 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.

The SSH hostname and port number is configured in the `etc/org.apache.karaf.shell.cfg` configuration file with the following default values:

```
sshPort=8101
sshHost=0.0.0.0
sshRealm=karaf
hostKey=${karaf.base}/etc/host.key
```

You can change this configuration using the config commands or by editing the above file, but you'll need to restart the ssh console in order for it to use the new parameters.

```
# define helper functions
bundle-by-sn = { bm = new java.util.HashMap ;  each (bundles) {
$bm put ($it symbolicName) $it } ; $bm get $1 }
bundle-id-by-sn = { b = (bundle-by-sn $1) ; if { $b } { $b
bundleId } { -1 } }
# edit config
config:edit org.apache.karaf.shell
config:propset sshPort 8102
config:update
# force a restart
osgi:restart --force (bundle-id-by-sn org.apache.karaf.shell.ssh)
```

CONNECTING AND DISCONNECTING REMOTELY

Using the `ssh:ssh` command

You can connect to a remote Karaf's console using the `ssh:ssh` command.

```
karaf@root> ssh:ssh -l karaf -P karaf -p 8101 hostname
```

> The default password is `karaf` but we recommend changing it. See the
> security section for more informations.

To confirm that you have connected to the correct Karaf instance, type `shell:info` at
the `karaf>` prompt. Information about the currently connected instance is returned, as
shown.

```
Karaf
  Karaf home                /local/apache-karaf-2.0.0
  Karaf base                /local/apache-karaf-2.0.0
  OSGi Framework            org.eclipse.osgi -
3.5.1.R35x_v20090827
JVM
  Java Virtual Machine      Java HotSpot(TM) Server VM version
14.1-b02
  ...
```

Using the Karaf client

The Karaf client allows you to securely connect to a remote Karaf instance without
having to launch a Karaf instance locally.

   For example, to quickly connect to a Karaf instance running in server mode on the
same machine, run the following command:

```
bin/client
```

More commonly, you would provide a hostname, port, username and password to
connect to a remote instance. And, if you were using the client within a larger script, you
could append console commands as follows:

```
bin/client -a 8101 -h hostname -u karaf -p karaf
features:install wrapper
```

To display the available options for the client, type:

```
> 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
  -p [password] specify the password
```

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

Using a plain SSH client

You can also connect using a plain SSH client from your *nix system or Windows SSH client like Putty.

```
~$ ssh -p 8101 karaf@localhost
karaf@localhost's password:
```

Disconnecting from a remote console

To disconnect from a remote console, press `Ctrl+D`, `shell:logout` or simply `logout` at the Karaf prompt.

STOPPING A REMOTE INSTANCE

Using the remote console

If you have connected to a remote console using the `ssh:ssh` command or the Karaf client, you can stop the remote instance using the `osgi:shutdown` command.

> Pressing `Ctrl+D` in a remote console simply closes the remote connection and returns you to the local shell.

Using the Karaf client

To stop a remote instance using the Karaf client, run the following from the `lib` directory:

```
bin/client -u karaf -p karaf -a 8101 hostname osgi:shutdown
```

Web console

The Karaf web console provides a graphical overview of the runtime.
You can use it to:

- install and uninstall features
- start, stop, install bundles
- create child instances
- configure Karaf
- view logging informations

INSTALLING THE WEB CONSOLE

The web console is not installed by default. To install it, run the following command from the Karaf prompt:

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

ACCESSING THE WEB CONSOLE

To access the console for an instance of Karaf running locally, enter the following address in your web browser:

```
http://localhost:8181/system/console
```

Log in with the username `karaf` and the password `karaf`. If you have changed the default user or password, use the one you have configured.

CHANGING THE WEB CONSOLE PORT NUMBER

By default, the console runs on port 8181. You can change the port number by creating the properties file, `etc/org.ops4j.pax.web.cfg`, and adding the following property setting (changing the port number to whatever value desired):

```
org.osgi.service.http.port=8181
```

Managing child instances

A child instance of Karaf is a copy that you can launch separately and deploy applications into. An instance does not contain the full copy of Karaf, but only a copy of the configuration files and data folder which contains all the runtime information, logs and temporary files.

USING THE ADMIN CONSOLE COMMANDS

The **admin** console commands allow you to create and manage instances of Karaf on the same machine. Each new runtime is a child instance of the runtime that created it. You can easily manage the children using names instead of network addresses. For details on the **admin** commands, see the `admin` commands.

CREATING CHILD INSTANCES

You create a new runtime instance by typing `admin:create` in the Karaf console.

As shown in the following example, `admin:create` causes the runtime to create a new runtime installation in the active runtime's `instances/name` directory. The new instance is a new Karaf instance and is assigned an SSH port number based on an incremental count starting at 8101 and a RMI registry port number based on an incremental count starting at 1099/44444.

```
karaf@root> admin:create alpha
Creating new instance on SSH port 8102 and RMI ports 1100/44445
at: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha
Creating dir:  /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
bin
Creating dir:  /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc
Creating dir:  /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
system
Creating dir:  /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
deploy
Creating dir:  /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
data
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/config.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
```

```
etc/jre.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/custom.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/java.util.logging.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.apache.felix.fileinstall-deploy.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.apache.karaf.log.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.apache.karaf.features.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.ops4j.pax.logging.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.ops4j.pax.url.mvn.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/startup.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/users.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/system.properties
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.apache.karaf.shell.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
etc/org.apache.karaf.management.cfg
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
bin/karaf
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
bin/start
Creating file: /x3/karaf/test/apache-karaf-2.2.4/instances/alpha/
bin/stop
karaf@root>
```

CHANGING A CHILD'S PORTS

You can change the SSH port number assigned to a child instance using the
`admin:change-ssh-port` command. The syntax for the command is:

```
admin:change-ssh-port instance port
```

Note that the child instance has to be stopped in order to run this command.

In the same way, you can change the RMI registry port number assigned to a child instance using the `admin:change-rmi-registry-port` command. The syntax for the command is:

```
admin:change-rmi-registry-port instance port
```

Note that the child instance has to be stopped in order to run this command.

STARTING CHILD INSTANCES

New instances are created in a stopped state. To start a child instance and make it ready to host applications, use the `admin:start` command. This command takes a single argument `instance-name` that identifies the child you want started.

LISTING ALL CONTAINER INSTANCES

To see a list of all Karaf instances running under a particular installation, use the `admin:list` command.

```
karaf@root>admin:list
  SSH Port    RMI Port    State        Pid  Name
[    8107] [      1106] [Started ] [10628] harry
[    8101] [      1099] [Started ] [20076] root
[    8106] [      1105] [Stopped ] [15924] dick
[    8105] [      1104] [Started ] [18224] tom
karaf@root>
```

CONNECTING TO A CHILD INSTANCE

You can connect to a started child instance's remote console using the `admin:connect` command which takes three arguments:

```
admin:connect [-u username] instance [command]
```

NB: the password will be prompted.

NB: you can directly execute shell command on the target instance.

Once you are connected to the child instance, the Karaf prompt changes to display the name of the current instance, as shown:

```
karaf@harry>
```

STOPPING A CHILD INSTANCE

To stop a child instance from within the instance itself, type `osgi:shutdown` or simply `shutdown`.

To stop a child instance remotely, in other words, from a parent or sibling instance, use the `admin:stop`:

```
admin:stop instance
```

DESTROYING A CHILD INSTANCE

You can permanently delete a stopped child instance using the `admin:destroy` command:

```
admin:destroy instance
```

Note that only stopped instances can be destroyed.

USING THE ADMIN SCRIPT

You can also manage the local instances of Karaf. The `admin` script in the `bin` directory provides the same commands as the `admin` console commands, apart from `admin:connect`.

```
bin/admin
Available commands:
  change-ssh-port - Changes the secure shell port of an existing
container instance.
  change-rmi-registry-port - Changes the RMI registry port (used
by management layer) of an existing container instance.
  create - Creates a new container instance.
  destroy - Destroys an existing container instance.
  list - List all existing container instances.
  start - Starts an existing container instance.
  stop - Stops an existing container instance.
Type 'command --help' for more help on the specified command.
```

For example, to list all the instances of Karaf on the local machine, type:

```
bin/admin list
```

Security

MANAGING AUTHENTICATION BY USERS AND PASSWORDS

The default security configuration uses a property file located at `etc/users.properties` to store authorized users and their passwords.

The default user name is `karaf` and the associated password is `karaf` too. We strongly encourage you to change the default password by editing the above file before moving Karaf into production.

The users are currently used in three different places in Karaf:
- access to the SSH console
- access to the JMX management layer
- access to the Web console
  Those three ways all delegate to the same JAAS based security authentication.

  The `users.properties` file contains one or more lines, each line defining a user, its password and the associated roles.

  ```
  user=password[,role][,role]...
  ```

MANAGING AUTHENTICATION BY KEY

For the SSH layer, Karaf supports the authentication by key, allowing to login without providing the password.

The SSH client (so bin/client provided by Karaf itself, or any ssh client like OpenSSH) uses a public/private keys pair that
will identify himself on Karaf SSHD (server side).

The keys allowed to connect are stored in `etc/keys.properties` file, following the format:

```
user=key,role
```

By default, Karaf allows a key for the karaf user:

```
# karaf=AAAAB3NzaC1kc3MAAACBAP1/
U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1AIlH7WT2NWPq/
xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/
63xhv4O1fnxqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPE
C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfBpKLZl6Ae1UlZAFMO/
```

```
7PSSoAAACBAKKSU2PFl/
qOLxIwmBZPPIcJshVe7bVUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/
53JjTuyk31drV2qxhIOsLDC9dGCWj47Y7TyhPdXh/
0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```

The easiest way to create key pair is to use OpenSSH.

You can create a key pair using:

```
ssh-keygen -t dsa -f karaf.id_dsa -N karaf
```

You have now the public and private keys:

```
-rw-------  1 jbonofre jbonofre    771 Jul 25 22:05 karaf.id_dsa
-rw-r--r--  1 jbonofre jbonofre    607 Jul 25 22:05
karaf.id_dsa.pub
```

You can copy in the content of the `karaf.id_dsa.pub` file in the `etc/keys.properties`:

```
karaf=AAAAB3NzaC1kc3MAAACBAJLj9vnEhu3/
Q9Cvym2jRDaNWkATgQiHZxmErCmiLRuD5Klfv+HT/+8WoYdnvj0YaXFP80phYhzZ7fbIO2LRFhY
vczoX/
8I3FziEfss07Hj1NAAAAFQD1dKEzkt4e7rBPDokPOMZigBh4kwAAAIEAiLnpbGNbKm8SNLUEc/
fJFswg4G4VjjngjbPZAjhkYe4+H2uYmynry6V+GOTS2kaFQGZRf9XhSpSwfdxKtx7vCCaoH9bZ6
Sww8f2stpitW2Oq7V7lDdDG81+N/D7/
rKDD5PjUyMsVqc1n9wCTmfqmi6XPEw8AAACAHAGwPn/
Mv7P9Q9+JZRWtGq+i4pL1zs1OluiStCN9e/
Ok96t3gRVKPheQ6IwLacNjC9KkSKrLtsVyepGA+V5j/N+Cmsl6csZilnLvMUTvL/
cmHDEEhTIQnPNrDDv+tED2BFqkajQqYLgMWeGVqXsBU6IT66itZlYtrq4v6uDQG/
o=,admin
```

and specify to the client to use the `karaf.id_dsa` private key:

```
bin/client -k ~/karaf.id_dsa
```

or to ssh

```
ssh -p 8101 -i ~/karaf.id_dsa karaf@localhost
```

MANAGING ROLES

JAAS roles can be used by various components. The three management layers (SSH, JMX and WebConsole) all use a global role based authorization system. The default role name is configured in the `etc/system.properties` using the `karaf.admin.role` system property and the default value is `admin`. All users authenticating for the management layer must have this role defined.

The syntax for this value is the following:

```
[classname:]principal
```

where classname is the class name of the principal object (defaults to org.apache.karaf.jaas.modules.RolePrincipal) and principal is the name of the principal of that class (defaults to admin).

Note that roles can be changed for a given layer using ConfigAdmin in the following configurations:

Layer	PID	Value
SSH	org.apache.karaf.shell	sshRole
JMX	org.apache.karaf.management	jmxRole
Web	org.apache.karaf.webconsole	role

ENABLING PASSWORD ENCRYPTION

In order to not keep the passwords in plain text, the passwords can be stored encrypted in the configuration file.
This can be easily enabled using the following commands:

```
# edit config
config:edit org.apache.karaf.jaas
config:propset encryption.enabled true
config:update
# force a restart
dev:restart
```

The passwords will be encrypted automatically in the `etc/users.properties` configuration file the first time the user logs in.
Encrypted passwords are prepended with `{CRYPT}` so that are easy to recognize.

MANAGING REALMS

More information about modifying the default realm or deploying new realms is provided in the developers guide.

DEPLOYING SECURITY PROVIDERS

Some applications require specific security providers to be available, such as BouncyCastle. The JVM impose some restrictions about the use of such jars: they have to be signed and be available on the boot classpath. One way to deploy those providers is to put them in the JRE folder at `$JAVA_HOME/jre/lib/ext` and modify the security policy configuration (`$JAVA_HOME/jre/lib/security/java.security`) in order to register such providers.

While this approach works fine, it has a global effect and requires you to configure all your servers accordingly.

Karaf offers a simple way to configure additional security providers:
- put your provider jar in `lib/ext`
- modify the `etc/config.properties` configuration file to add the following property

```
org.apache.karaf.security.providers = xxx,yyy
```

The value of this property is a comma separated list of the provider class names to register.
For example:

```
org.apache.karaf.security.providers =
org.bouncycastle.jce.provider.BouncyCastleProvider
```

In addition, you may want to provide access to the classes from those providers from the system bundle so that all bundles can access those. It can be done by modifying the `org.osgi.framework.bootdelegation` property in the same configuration file:

```
org.osgi.framework.bootdelegation = ...,org.bouncycastle*
```

Failover Deployments

Karaf provides failover capability using either a simple lock file system or a JDBC locking mechanism. In both cases, a container-level lock system allows bundles to be preloaded into the slave Karaf instance in order to provide faster failover performance.

SIMPLE LOCK FILE

The simple lock file mechanism is intended for failover configurations where instances reside on the same host machine.

To use this feature, edit the `$KARAF_HOME/etc/system.properties` file as follows on each system in the master/slave setup:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.SimpleFileLock
karaf.lock.dir=<PathToLockFileDirectory>
karaf.lock.delay=10000
```

Note: Ensure that the `karaf.lock.dir` property points to the same directory for both the master and slave instance, so that the slave can acquire the lock only when the master releases it.

JDBC LOCKING

The JDBC locking mechanism is intended for failover configurations where instances exist on separate machines. In this deployment, the master instance holds a lock on a Karaf locking table hosted on a database. If the master loses the lock, a waiting slave process gains access to the locking table and fully starts its container.

To use this feature, do the following on each system in the master/slave setup:
- Update the classpath to include the JDBC driver
- Update the `$KARAF_HOME/bin/karaf` script to have a unique JMX remote port set if instances reside on the same host
- Update the `$KARAF_HOME/etc/system.properties` file as follows:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.DefaultJDBCLock
karaf.lock.level=50
karaf.lock.delay=10000
karaf.lock.jdbc.url=jdbc:derby://dbserver:1527/sample
```

```
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

Note:

- This process will fail if a JDBC driver is not on the classpath.
- The "sample" database referred to above will be created if it does not exist.
- The first Karaf instance to acquire the locking table is the master instance.
- If the connection to the database is lost, the master instance tries to gracefully shutdown, allowing a slave instance to become master when the database service is restored. The former master will require a manual restart.
- Karaf will first use the table name as defined in karaf.lock.jdbc.table property, it will also try upper and lower case for the table name.

> Apache Karaf won't start if the JDBC driver is not present in the `lib/ext` folder.

JDBC locking on Oracle

If you are using Oracle as your database for JDBC locking, the `karaf.lock.class` property in the `$KARAF_HOME/etc/system.properties` file must point to `org.apache.karaf.main.OracleJDBCLock`.

Otherwise, configure the system.properties file as normal for your setup, for example:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.OracleJDBCLock
karaf.lock.jdbc.url=jdbc:oracle:thin:@hostname:1521:XE
karaf.lock.jdbc.driver=oracle.jdbc.OracleDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

Make sure you have the Oracle JDBC driver in the `lib/ext` folder.

Note: The `karaf.lock.jdbc.url` requires an active SID, which means you must manually create a database instance before using this particular lock.

Derby

Make sure you have the driver jar file in the Karaf `lib/ext` folder.

Then make you update the properties in `$KARAF_HOME/etc/system.properties` to look something like this example:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.DerbyJDBCLock
karaf.lock.jdbc.url=jdbc:derby://127.0.0.1:1527/dbname
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

MySQL

Make sure you have the MySQL driver jar file in the Karaf `lib/ext` folder.

Then make you update the properties in `$KARAF_HOME/etc/system.properties` to look something like this example:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.MySQLJDBCLock
karaf.lock.jdbc.url=jdbc:mysql://127.0.0.1:3306/dbname
karaf.lock.jdbc.driver=com.mysql.jdbc.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

PostgreSQL

Make sure you have the PostgreSQL driver jar file in the Karaf `lib/ext` folder.

Then make you update the properties in `$KARAF_HOME/etc/system.properties` to look something like this example:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.PostgreSQLJDBCLock
karaf.lock.jdbc.url=jdbc:postgresql://127.0.0.1:5432/dbname
karaf.lock.jdbc.driver=org.postgresql.Driver
karaf.lock.jdbc.user=user
```

```
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=0
```

Microsoft SQLServer

Make sure you have the JTDS JDBC driver jar file in the Karaf `lib/ext` folder.

Then make you update the properties in `KARAF_HOME/etc/system.properties` to look something like this example:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.lock.SQLServerJDBCLock
karaf.lock.level=50
karaf.lock.delay=10
karaf.lock.jdbc.url=jdbc:jtds:sqlserver://127.0.0.1;databaseName=sample
karaf.lock.jdbc.driver=net.sourceforge.jtds.jdbc.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

CONTAINER-LEVEL LOCKING

Container-level locking allows bundles to be preloaded into the slave kernel instance in order to provide faster failover performance. Container-level locking is supported in both the simple file and JDBC locking mechanisms.

To implement container-level locking, add the following to the `$KARAF_HOME/etc/system.properties` file on each system in the master/slave setup:

```
karaf.lock=true
karaf.lock.level=50
karaf.lock.delay=10000
```

The `karaf.lock.level` property tells the Karaf instance how far into the boot process to bring the OSGi container. All bundles with an ID equal or lower to this start level will be started in that Karaf instance.

Bundle start levels are specified in `$KARAF_HOME/etc/startup.properties`, in the format `jar.name=level`. The core system bundles have levels below 50, where user bundles have levels greater than 50.

Level	Behavior

1	A 'cold' standby instance. Core bundles are not loaded into container. Slaves will wait until lock acquired to start server.
<50	A 'hot' standby instance. Core bundles are loaded into the container. Slaves will wait until lock acquired to start user level bundles. The console will be accessible for each slave instance at this level.
>50	This setting is not recommended as user bundles will end up being started.

Note: When using a 'hot' spare on the same host you need to set the JMX remote port to a unique value to avoid bind conflicts. You can edit the Karaf start script to include the following:

```
DEFAULT_JAVA_OPTS="-server $DEFAULT_JAVA_OPTS
-Dcom.sun.management.jmxremote.port=1100
-Dcom.sun.management.jmxremote.authenticate=false"
```

Logging system

Karaf provides a powerful logging system based on OPS4j Pax Logging.

In addition to being a standard OSGi Log service, it supports the following APIs:
- Apache Commons Logging
- SLF4J
- Apache Log4j
- Java Util Logging

Karaf also comes with a set of console commands that can be used to display, view and change the log levels.

CONFIGURATION

Configuration file

The configuration of the logging system uses a standard Log4j configuration file at the following location:

```
[karaf_install_dir]/etc/org.ops4j.pax.logging.cfg
```

You can edit this file at runtime and any change will be reloaded and be effective immediately.

Configuring the appenders

The default logging configuration defines three appenders:
- the `stdout` console appender is disabled by default. If you plan to run Karaf in server mode only (i.e. with the locale console disabled), you can turn on this appender on by adding it to the list of configured appenders using the `log4j.rootLogger` property
- the `out` appender is the one enabled by default. It logs events to a number of rotating log files of a fixed size. You can easily change the parameters to control the number of files using `maxBackupIndex` and their size size `maxFileSize`
- the `sift` appender can be used instead to provide a per-bundle log file. The default configuration uses the bundle symbolic name as the file name to log to

Changing the log levels

The default logging configuration sets the logging levels so that the log file will provide enough information to monitor the behavior of the runtime and provide clues about what caused a problem. However, the default configuration will not provide enough information to debug most problems.

The most useful logger to change when trying to debug an issue with Karaf is the root logger. You will want to set its logging level to `DEBUG` in the `org.ops4j.pax.logging.cfg` file.

```
log4j.rootLogger=DEBUG, out, osgi:VmLogAppender
...
```

When debugging a problem in Karaf you may want to change the level of logging information that is displayed on the console. The example below shows how to set the root logger to `DEBUG` but limiting the information displayed on the console to WARN.

```
log4j.rootLogger=DEBUG, out, stdout, osgi:VmLogAppender
log4j.appender.stdout.threshold=WARN
...
```

CONSOLE LOG COMMANDS

The log subshell comes with the following commands:
- `log:clear`: clear the log
- `log:display`: display the last log entries
- `log:display-exception`: display the last exception from the log
- `log:get`: show the log levels. `log:get ALL` displays the level of all loggers.
- `log:list`: is an alias to `log:get ALL`.
- `log:set`: set the log levels. NB: this command doesn't update `etc/org.ops4j.pax.logging.cfg` file. You have to use `config:edit` to do so. It's possible to reload the log level from the configuration file using `log:set DEFAULT`.
- `log:tail`: continuous display of the log entries

For example, if you want to debug something, you might want to run the following commands:

```
> log:set DEBUG
... do something ...
> log:display
```

Note that the log levels set using the `log:set` commands are not persistent and will be lost upon restart.

To configure those in a persistent way, you should edit the configuration file mentioned above using the config commands or directly using a text editor of your choice.

The log commands has a separate configure file:

```
[karaf_install_dir]/etc/org.apache.karaf.log.cfg
```

ADVANCED CONFIGURATION

The logging backend uses Log4j, but offer a number of additional features.

Nested filters, appenders and error handlers

Filters

Appender filters can be added using the following syntax:

```
log4j.appender.[appender-name].filter.[filter-name]=[filter-class]
log4j.appender.[appender-name].filter.[filter-name].[option]=[value]
```

Below is a real example:

```
log4j.appender.out.filter.f1=org.apache.log4j.varia.LevelRangeFilter
log4j.appender.out.filter.f1.LevelMax=FATAL
log4j.appender.out.filter.f1.LevelMin=DEBUG
```

Nested appenders

Nested appenders can be added using the following syntax:

```
log4j.appender.[appender-name].appenders=[comma-separated-list-of-appender-names]
```

Below is a real example:

```
log4j.appender.async=org.apache.log4j.AsyncAppender
log4j.appender.async.appenders=jms
```

```
log4j.appender.jms=org.apache.log4j.net.JMSAppender
...
```

Error handlers

Error handlers can be added using the following syntax:

```
log4j.appender.[appender-name].errorhandler=[error-handler-class]
log4j.appender.[appender-name].errorhandler.root-ref=[true|false]
log4j.appender.[appender-name].errorhandler.logger-ref=[logger-ref]
log4j.appender.[appender-name].errorhandler.appender-ref=[appender-ref]
```

OSGi specific MDC attributes

Pax-Logging provides the following attributes by default:
- `bundle.id`: the id of the bundle from which the class is loaded
- `bundle.name`: the symbolic-name of the bundle
- `bundle.version`: the version of the bundle

An MDC sifting appender is available to split the log events based on MDC attributes.
Below is a configuration example for this appender:

```
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=karaf
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\\}.log
log4j.appender.sift.appender.append=true
```

Enhanced OSGi stack trace renderer

This renderer is configured by default in Karaf and will give additional informations when printing stack traces.
For each line of the stack trace, it will display OSGi specific informations related to the class on that line: the bundle id, the bundle symbolic name and the bundle version. This information can greatly help diagnosing problems in some cases.

The information is appended at the end of each line in the following format
`id:name:version` as shown below

```
java.lang.IllegalArgumentException: Command not found:  *:foo
        at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:225)[21:org.apac
        at
org.apache.felix.gogo.runtime.shell.Closure.executeStatement(Closure.java:162)[21
        at
org.apache.felix.gogo.runtime.shell.Pipe.run(Pipe.java:101)[21:org.apache.karaf.s
        at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:79)[21:org.apach
        at
org.apache.felix.gogo.runtime.shell.CommandSessionImpl.execute(CommandSessionImpl
        at
org.apache.karaf.shell.console.jline.Console.run(Console.java:169)[21:org.apache.
        at java.lang.Thread.run(Thread.java:637)[:1.6.0_20]
```

Using your own appenders

If you plan to use your own appenders, you need to create an OSGi bundle and attach it
as a fragment to the bundle with a symbolic name of
`org.ops4j.pax.logging.pax-logging-service`. This way, the underlying
logging system will be able to see and use your appenders.

So for example you write a log4j appender:
class MyAppender extends AppenderSkeleton {

...

}

Then you need to package the appender in a jar with a Manifest like this:

Manifest:

Bundle-SymbolicName: org.mydomain.myappender
Fragment-Host: org.ops4j.pax.logging.pax-logging-service

...

Copy the new appender fragment into the ${karaf.home}/system directory. Karaf
uses a virtual maven repository to access resources in the system
directory, so the jar path should use the standard maven groupId/artifactId/version/
convention, where the groupId is a directory structure.

Edit ${karaf.home}/etc/startup.properties and add the new fragment bundle to the list
before the pax-logging-service bundle.

Restart karaf with a clean run to reload the system bundles, and now you can use the
appender in your log4j config file like shown in the config
examples above.

Deployer

The following picture describes the architecture of the deployer.

BLUEPRINT DEPLOYER

Karaf includes a deployer that is able to deploy plain blueprint configuration files. The deployer will transform on the fly any spring configuration file dropped into the `deploy` folder into a valid OSGi bundle.

The generated OSGi manifest will contain the following headers:

```
Manifest-Version: 2
Bundle-SymbolicName: [name of the file]
Bundle-Version: [version of the file]
Import-Package: [required packages]
DynamicImport-Package: *
```

The `name` and `version` of the file are extracted using a heuristic that will match common patterns. For example `my-config-1.0.1.xml` will lead to `name = my-config` and `version = 1.0.1`.
The default imported packages are extracted from the spring file definition and includes all classes referenced directly.

If you need to customize the generated manifest, you can do so by including an xml element in your blueprint configuration:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <manifest xmlns="http://karaf.apache.org/xmlns/deployer/
blueprint/v1.0.0">
    Require-Bundle= my-bundle
  </manifest>
```

SPRING DEPLOYER

Similar to the blueprint deployer, you can deploy spring-dm configuration files.

The generated OSGi manifest will contain the following headers:

```
Manifest-Version: 2
Bundle-SymbolicName: [name of the file]
Bundle-Version: [version of the file]
Spring-Context:
*;publish-context:=false;create-asynchronously:=true
Import-Package: [required packages]
DynamicImport-Package: *
```

If you need to customize the generated manifest, you can do so by including an xml element in your spring configuration:

```
<spring:beans ...>
  <manifest xmlns="http://karaf.apache.org/xmlns/deployer/spring/
v1.0.0">
    Require-Bundle= my-bundle
  </manifest>
```

FEATURES DEPLOYER

To be able to hot deploy features from the deploy folder, you can just drop a feature descriptor on that folder. A bundle will be created and its installation (automatic) will trigger the installation of all features contained in the descriptor. Removing the file from the deploy folder will uninstall the features.
If you want to install a single feature, you can do so by writing a feature descriptor like the following:

```
<features>
  <repository>mvn:org.apache.servicemix.nmr/
apache-servicemix-nmr/1.0.0/xml/features</repository>
  <feature name="nmr-only">
    <feature>nmr</feature>
  </feature>
</features>
```

For more informations about features, see the provisioning section.

WAR DEPLOYER

To be able to hot deploy web application (war) from the deploy folder, you have to install the war feature:

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

NB: you can use the -v or --verbose option to see exactly what is performed by the feature deployer.

```
karaf@root> features:install -v war
Installing feature war 2.1.99-SNAPSHOT
Installing feature http 2.1.99-SNAPSHOT
Installing feature jetty 7.2.2.v20101205
Installing bundle mvn:org.apache.geronimo.specs/
geronimo-servlet_2.5_spec/1.1.2
Found installed bundle: org.apache.servicemix.bundles.asm [9]
Installing bundle mvn:org.eclipse.jetty/jetty-util/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-io/7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-http/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-continuation/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-server/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-security/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-servlet/
7.2.2.v20101205
Installing bundle mvn:org.eclipse.jetty/jetty-xml/7.2.2.v20101205
Checking configuration file mvn:org.apache.karaf/apache-karaf/
2.1.99-SNAPSHOT/xml/jettyconfig
Installing bundle mvn:org.ops4j.pax.web/pax-web-api/1.0.0
Installing bundle mvn:org.ops4j.pax.web/pax-web-spi/1.0.0
Installing bundle mvn:org.ops4j.pax.web/pax-web-runtime/1.0.0
Installing bundle mvn:org.ops4j.pax.web/pax-web-jetty/1.0.0
Installing bundle mvn:org.apache.karaf.shell/
org.apache.karaf.shell.web/2.1.99-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-jsp/1.0.0
Installing bundle mvn:org.ops4j.pax.web/pax-web-extender-war/
1.0.0
Installing bundle mvn:org.ops4j.pax.web/
pax-web-extender-whiteboard/1.0.0
Installing bundle mvn:org.ops4j.pax.web/pax-web-deployer/1.0.0
Installing bundle mvn:org.ops4j.pax.url/pax-url-war/1.2.5
```

As you can see, the war feature uses PAX Web as war deployer.

You should now be able to see the PAX Web war deployer:

```
karaf@root> osgi:list |grep -i war
[  57] [Active     ] [              ] [   60] OPS4J Pax Web -
Extender - WAR (1.0.0)
[  60] [Active     ] [              ] [   60] OPS4J Pax Url -
war:, war-i: (1.2.5)
```

You can deploy a web application packaged in war or exploded in a directory.

Your web application should at least contain a WEB-INF/web.xml file.

WRAP DEPLOYER

The wrap deployer allows you to hot deploy non-OSGi jar files ("classical" jar files) from the deploy folder.

It's a standard deployer (you don't need to install additional Karaf features):

```
karaf@root> la|grep -i wrap
[   1] [Active     ] [              ] [    5] OPS4J Pax Url -
wrap: (1.2.5)
[  32] [Active     ] [Created      ] [   30] Apache Karaf ::
Deployer :: Wrap Non OSGi Jar (2.1.99.SNAPSHOT)
```

Karaf wrap deployer looks for jar files in the deploy folder. The jar files is considered as non-OSGi if the MANIFEST
doesn't contain the Bundle-SymbolicName and Bundle-Version attributes, or if there is no MANIFEST at all.

The non-OSGi jar file is transformed into an OSGi bundle.

The deployer tries to populate the Bundle-SymbolicName and Bundle-Version extracted from the jar file path.

For example, if you simply copy commons-lang-2.3.jar (which is not an OSGi bundle) into the deploy folder, you
will see:

```
karaf@root> la|grep -i commons-lang
[  41] [Active     ] [              ] [   60] commons-lang (2.3)
```

If you take a look on the commons-lang headers, you can see that the bundle exports all packages with optional resolution
and that Bundle-SymbolicName and Bundle-Version have been populated:

```
karaf@root> osgi:headers 41

commons-lang (41)
-----------------
```

```
Specification-Title = Commons Lang
Tool = Bnd-0.0.357
Specification-Version = 2.3
Specification-Vendor = Apache Software Foundation
Implementation-Version = 2.3
Generated-By-Ops4j-Pax-From = wrap:file:/home/onofreje/workspace/
karaf/assembly/target/apache-karaf-2.99.99-SNAPSHOT/deploy/
commons-lang-2.3.jar$Bundle-SymbolicName=commons-lang&Bundle-Version=2.3
Implementation-Vendor-Id = org.apache
Created-By = 1.6.0_21 (Sun Microsystems Inc.)
Implementation-Title = Commons Lang
Manifest-Version = 1.0
Bnd-LastModified = 1297248243231
X-Compile-Target-JDK = 1.1
Originally-Created-By = 1.3.1_09-85 ("Apple Computer, Inc.")
Ant-Version = Apache Ant 1.6.5
Package = org.apache.commons.lang
X-Compile-Source-JDK = 1.3
Extension-Name = commons-lang
Implementation-Vendor = Apache Software Foundation

Bundle-Name = commons-lang
Bundle-SymbolicName = commons-lang
Bundle-Version = 2.3
Bundle-ManifestVersion = 2

Import-Package =
        org.apache.commons.lang;resolution:=optional,
        org.apache.commons.lang.builder;resolution:=optional,
        org.apache.commons.lang.enum;resolution:=optional,
        org.apache.commons.lang.enums;resolution:=optional,
        org.apache.commons.lang.exception;resolution:=optional,
        org.apache.commons.lang.math;resolution:=optional,
        org.apache.commons.lang.mutable;resolution:=optional,
        org.apache.commons.lang.text;resolution:=optional,
        org.apache.commons.lang.time;resolution:=optional
Export-Package =

org.apache.commons.lang;uses:="org.apache.commons.lang.builder,org.apache.c

org.apache.commons.lang.builder;uses:="org.apache.commons.lang.math,org.apa

org.apache.commons.lang.enum;uses:=org.apache.commons.lang,
```

```
org.apache.commons.lang.enums;uses:=org.apache.commons.lang,

org.apache.commons.lang.exception;uses:=org.apache.commons.lang,

org.apache.commons.lang.math;uses:=org.apache.commons.lang,

org.apache.commons.lang.mutable;uses:="org.apache.commons.lang,org.apache.c

org.apache.commons.lang.text;uses:=org.apache.commons.lang,

org.apache.commons.lang.time;uses:=org.apache.commons.lang
```

You may set the manifest headers by specifying them as URL parameters, '$' is used
separate the hierarch path and query parts of the URL, with standard URL syntax for
query parameters of key=value pairs joined by '&'.

On the command line you must use single quotes around the URL to prevent the
shell from attempting variable expansion, and if present, you must escape exclamation
marks with a backslash.

```
install -s 'wrap:mvn:jboss/jbossall-client/
4.2.3.GA/$Bundle-SymbolicName=jbossall-client&Bundle-Version=4.2.3.GA&Expor
```

When defined in a features.xml file, it's necessary to escape any ampersands and
quotes, or use a CDATA tag.

```
<bundle>wrap:mvn:jboss/jbossall-client/
4.3.2.GA/$Bundle-SymbolicName=jbossall-client&undle-Version=4.3.2.GA&xport-
```

Provisioning

Karaf provides a simple, yet flexible, way to provision applications or "features". Such a mechanism is mainly provided by a set of commands available in the `features` shell. The provisioning system uses xml "repositories" that define a set of features.

REPOSITORIES

The complete xml schema for feature descriptor are available on Features XML Schema page. We recommend using this XML schema. It will allow Karaf to validate your repository before parsing. You may also verify your descriptor before adding it to Karaf by simply validation, even from IDE level.

Here is an example of such a repository:

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.0.0">
    <feature name="spring" version="3.0.4.RELEASE">
        <bundle>mvn:org.apache.servicemix.bundles/
org.apache.servicemix.bundles.aopalliance/1.0_1</bundle>
        <bundle>mvn:org.springframework/spring-core/
3.0.4.RELEASE</bundle>
        <bundle>mvn:org.springframework/spring-beans/
3.0.4.RELEASE</bundle>
        <bundle>mvn:org.springframework/spring-aop/
3.0.4.RELEASE</bundle>
        <bundle>mvn:org.springframework/spring-context/
3.0.4.RELEASE</bundle>
        <bundle>mvn:org.springframework/spring-context-support/
3.0.4.RELEASE</bundle>
    </feature>
</features>
```

A repository includes a list of `feature` elements, each one representing an application that can be installed. The feature is identified by its `name` which must be unique amongst all the repositories used and consists of a set of bundles that need to be installed along with some optional dependencies on other features and some optional configurations for the Configuration Admin OSGi service.

References to features define in other repositories are allow and can be achieved by adding a list of repository.

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.0.0">
  <repository>mvn:org.apache.servicemix.nmr/
apache-servicemix-nmr/1.3.0/xml/features</repository>
  <repository>mvn:org.apache.camel.karaf/apache-camel/2.5.0/xml/
features</repository>
  <repository>mvn:org.apache.karaf/apache-karaf/2.1.2/xml/
features</repository>
  ...
```

> Be careful when you define them as there is a risk of 'cycling' dependencies.

Remark: By default, all the features defined in a repository are not installed at the launch of Apache Karaf (see section hereafter 'h2. Service configuration' for more info).

Bundles

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.

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. This flexible tool can be configured through the configuration service. For example, to find the current repositories type:

```
karaf@root:/> config:list
...
----------------------------------------------------------------
Pid:            org.ops4j.pax.url.mvn
BundleLocation: mvn:org.ops4j.pax.url/pax-url-mvn/0.3.3
Properties:
   service.pid = org.ops4j.pax.url.mvn
   org.ops4j.pax.url.mvn.defaultRepositories = file:/opt/
development/karaf/assembly/target/
apache-felix-karaf-1.2.0-SNAPSHOT/system@snapshots
   org.ops4j.pax.url.mvn.repositories = http://repo1.maven.org/
maven2,
                                      http://svn.apache.org/
repos/asf/servicemix/m2-repo
   below = list of repositories and even before the local
repository
```

The repositories checked are controlled by these configuration properties.

For example, `org.ops4j.pax.url.mvn.repositories` is a comma separate list of repository URLs specifying those remote repositories to be checked. So, to replace the defaults with a new repository at `http://www.example.org/repo` on the local machine:

```
karaf@root:/> config:edit org.ops4j.pax.url.mvn
karaf@root:/> config:proplist
   service.pid = org.ops4j.pax.url.mvn
   org.ops4j.pax.url.mvn.defaultRepositories = file:/opt/
development/karaf/assembly/target/
apache-felix-karaf-1.2.0-SNAPSHOT/system@snapshots
   org.ops4j.pax.url.mvn.repositories = http://repo1.maven.org/
maven2,
                                      http://svn.apache.org/
repos/asf/servicemix/m2-repo
   below = list of repositories and even before the local
repository
karaf@root:/> config:propset org.ops4j.pax.url.mvn.repositories
http://www.example.org/repo
karaf@root:/> config:update
```

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

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

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

Bundle start-level

Available since Karaf 2.0

By default, the bundles deployed through the feature mechanism will have a start-level equals to the value defined in the configuration file `config.properties` with the variable `karaf.startlevel.bundle=80`. This value can be changed using the xml attribute start-level.

```
<feature name='my-project' version='1.0.0'>
  <feature version='2.4.0'>camel-spring</feature>
  <bundle start-level='80'>mvn:com.mycompany.myproject/
myproject-dao</bundle>
  <bundle start-level='85'>mvn:com.mycompany.myproject/
myproject-service</bundle>
  <bundle start-level='85'>mvn:com.mycompany.myproject/
myproject-camel-routing</bundle>
</feature>
```

The advantage in defining the bundle start-level is that you can deploy all your bundles including any required 'infrastructure' bundles (e.g Camel, ActiveMQ) at the same time and you will have the guarantee when using Spring Dynamic Modules or Blueprint that the Spring context will not be created without all the required services installed.

Bundle 'stop/start'

The OSGI specification allows for installing a bundle without starting it. To use this functionality, simply add the following attribute in your <bundle> definition

```
<feature name='my-project' version='1.0.0'>
  <feature version='2.4.0'>camel-spring</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>
  <bundle start-level='85'
start='false'>mvn:com.mycompany.myproject/
```

```
myproject-camel-routing</bundle>
  </feature>
```

> On Karaf 2.x, the start-level was not considered during the feature startup, but only the order in which bundles
> are defined in your feature.xml.
> Starting with Karaf 3.x, the start-level is considered correctly.
> If you need to use the old behavior you can uncomment and change the respectStartLvlDuringFeatureStartup variable
> in org.apache.karaf.features.xml to false. But please be aware that it will be removed in 4.0 and should therefore
> be used only temporarily.

Bundle 'dependency'

A bundle can be flagged as being a dependency. Such information can be used by resolvers to compute the full list of bundles to be installed.

Dependent features

Dependent features are useful when a given feature depends on another feature to be installed. Such a dependency can be expressed easily in the feature definition:

```
<feature name="jbi">
  <feature>nmr</feature>
  ...
</feature>
```

The effect of such a dependency is to automatically install the required `nmr` feature when the `jbi` feature is installed.

A version range can be specified on the feature dependency:

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

In such a case, if no matching feature is already installed, 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.

Configurations

The configuration section allows for declaring deployment configuration of the OSGi Configuration Admin service along a set of bundles.
Here is an example of such a configuration:

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

The `name` attribute of the `configuration` element will be used as the ManagedService PID for the configuration set in the Configuration Admin service. When using a ManagedServiceFactory, the `name` attribute is *servicePid*-_aliasId_, where *servicePid* is the PID of the ManagedServiceFactory and *aliasId* is a label used to uniquely identify a particular service (an alias to the factory generated service PID).

   Deploying such a configuration has the same effect as dropping a file named `com.foo.bar.cfg` into the `etc` folder.

   The content of the `configuration` element is set of properties parsed using the standard java property mechanism.

   Such configuration as usually used with Spring-DM or Blueprint support for the Configuration Admin service, as in the following example, but using plain OSGi APIs will of course work the same way:

```
<bean ...>
    <property name="propertyName" value="${myProperty}" />
</bean>

<osgix:cm-properties id="cmProps" persistent-id="com.foo.bar">
    <prop key="myProperty">myValue</prop>
</osgix:cm-properties>
<ctx:property-placeholder properties-ref="cmProps" />
```

There may also be cases where you want to make the properties from multiple configuration files available to your bundle context. This is something you may want to do if you have a multi-bundle application where there are application properties used by multiple bundles, and each bundle has its own specific properties. In that case, `<ctx:property-placeholder>` won't work as it was designed to make only one configuration file available to a bundle context.
To make more than one configuration file available to your bundle-context you would do something like this:

```
<beans:bean id="myBundleConfigurer"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfig">
```

```
    <beans:property name="ignoreUnresolvablePlaceholders"
value="true"/>
    <beans:property name="propertiesArray">
        <osgix:cm-properties id="myAppProps"
persistent-id="myApp.props"/>
        <osgix:cm-properties id="myBundleProps"
persistent-id="my.bundle.props"/>
    </beans:property>
</beans:bean>
```

In this example, we are using SpringDM with osgi as the primary namespace. Instead of using ctx:context-placeholder we are using the "PropertyPlaceholderConfig"
class. Then we are passing in a beans array and inside of that array is where we set our osgix:cm-properties elements. This element "returns" a properties bean.

For more information about using the Configuration Admin service in Spring-DM, see the Spring-DM documentation.

Configuration files

In certain cases it is needed not only to provide configurations for the configuration admin service but to add additional
configuration files e.g. a configuration file for jetty (jetty.xml). It even might be helpful to deploy a configuration
file instead of a configuration for the config admin service since. To achieve this the attribute `finalname` shows the
final destination of the `configfile`, while the value references the Maven artifact to deploy.
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.

```
<configfile finalname="/etc/jetty.xml"
override="false">mvn:org.apache.karaf/apache-karaf/2.3.12/xml/
jettyconfig</configfile>
```


Feature resolver

The resolver attribute on a feature can be set to force the use of a given resolver instead of the default resolution process. A resolver will be use to obtain the list of bundles to actually install for a given feature.
The default resolver will simply return the list of bundles provided in the feature description.
The OBR resolver can be installed and used instead of the standard one. In that case,

the resolver will use the OBR service
to determine the list of bundles to install (bundles flagged as dependency will only be
used as possible candidates to solve
various constraints).

COMMANDS

Repository management

The following commands can be used to manage the list of descriptors known by Karaf.
They use URLs pointing to features descriptors. These URLs can use any protocol
known to the Apache Karaf, the most common ones being http, file and mvn.

```
features:addUrl      Add a list of repository URLs to the
features service
features:removeUrl   Remove a list of repository URLs from the
features service
features:listUrl     Display the repository URLs currently
associated with the features service.
features:refreshUrl  Reload the repositories to obtain a fresh
list of features
```

Karaf maintains a persistent list of these repositories so that if you add one URL and
restart Karaf, the features will still be available.

The `refreshUrl` command is mostly used when developing features descriptors:
when changing the descriptor, it can be
handy to reload it in the Kernel without having to restart it or to remove then add the
URL again.

Features management

```
features:install
features:uninstall
features:list
```

Examples

1. Install features using mvn handler

```
features:addUrl mvn:org.apache.servicemix.nmr/
apache-servicemix-nmr/1.0.0-m2/xml/features
features:install nmr
```

2. Use file handler to deploy features file

```
features:addUrl file:base/features/features.xml
```

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

3. 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

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.0.0">
    <feature name="spring-web" version="2.5.6.SEC01">
        <bundle>file:base/bundles/
spring-web-2.5.6.SEC01.jar</bundle>
    </feature>
</features>
```

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

SERVICE CONFIGURATION

A simple configuration file located in `etc/org.apache.karaf.features.cfg` can be modified to customize the behavior when starting the Kernel for the first time. This configuration file contains two properties:

- `featuresBoot`: a comma separated list of features to install at startup
- `featuresRepositories`: a comma separated list of feature repositories to load at startup

This configuration file is of interest if you plan to distribute a customized Karaf distribution having pre-installed features. Such a process is detailed in the building custom distributions section.

XML Schema for provisioning

The following schema can be found in Karaf sources. It is also available publicly at url http://karaf.apache.org/xmlns/features/v1.0.0.

Karaf Archives (KAR)

Karaf provides a specific archive format named the KAR (Karaf ARchive).

Basically, the kar format is a jar (so a zip file) which contains a set of feature descriptor and bundle jar files.

For instance, a kar looks like:

- my-features-1.xml
- bundle1.jar
- bundle2.jar
- bundle3.jar

all packaged in zip format.

CREATE A KAR ARCHIVE

You can create a kar file by hand, just by zip compressing a directory representing the kar content.

You can also use the Karaf features maven plugin. The features maven plugin provides an create-kar goal.

The kar-archive goal:

1. Reads all features specified in the features descriptor.
2. For each feature, it resolves the bundles defined in the feature.
3. All bundles are packaged into the kar archive.

For instance, you can use the following POM to create a kar:

```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.groupId</groupId>
    <artifactId>my-kar</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>

    <build>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>features-maven-plugin</artifactId>
                <version>2.2.8</version>
                <executions>
                    <execution>
                        <id>create-kar</id>
                        <goals>
                            <goal>create-kar</goal>
                        </goals>
                        <configuration>
                            <featuresFile>src/main/resources/
features.xml</featuresFile>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

For the example, the features descriptor is very simple:

```
<?xml version="1.0" encoding="UTF-8"?>
<features>

    <feature name="my" version="1.0">
        <bundle>mvn:commons-collections/commons-collections/
3.2.1</bundle>
    </feature>

</features>
```

To create the kar archive, simply type:

```
mvn install
```

and you will have your kar in the `target` directory.

DEPLOY A KAR ARCHIVE

Karaf provides a KAR deployer:

```
karaf@root> la|grep -i archive
[  12] [Active      ] [Created      ] [   30] Apache Karaf ::
Deployer :: Karaf Archive (.kar) (2.2.4)
```

It's a core deployer (you don't need to install additional features).

To deploy a kar, simply drop the kar into the deploy directory. The KAR Deployer will deploy all the kar content starting
from the features descriptor.

The KAR Deployer uncompress KAR archives in the system repository and register the features descriptor.
All features contained in a KAR archive will be automatically installed at deployment time.
You can now see your features installed:

```
karaf@root> features:list|grep -i my
[  installed] [1.0             ] my
repo-0
```

Configuration

The files in the `etc` directory are used to set the startup configuration.

For dynamic configuration, Karaf provides a suite of command to administer the configuration service grouped under `config`. To learn about all currently supported configuration commands type:

Command	Description
cancel	Change the changes to the configuration being edited.
edit	Create or edit a configuration.
list	List existing configurations.
propdel	Delete a property from the edited configuration.
proplist	List properties from the edited configuration.
propset	Set a property on the edited configuration.
update	Save and propagate changes from the configuration being edited.

EDITING

Select Configuration To Edit

For example to edit configuration `foo.bar`:

```
karaf@root> config:edit foo.bar
```

Modify Properties

Use:
   * `config:proplist` to list existing properties
* `config:propdel` to delete existing properties
* `config:propset` to set a new value for a property

Any number of properties can be modified within a single editing session.

Commit Or Rollback Changes

Use

* `config:update` to commit all changes made in the current session
* `config:cancel` to roll back any changes made in the current session

Pax Web (OSGi HttpService)

The Karaf http feature enables the Pax Web implementation of the OSGi HTTP service.

INSTALLING THE HTTP FEATURE

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

Create a file `etc/org.ops4j.pax.web.cfg` with the following content:

```
org.osgi.service.http.port=8080
```

This tells Pax Web to listen to the port 8080. As soon as the http feature is installed and the config
file is present, the following URL will be reachable:

```
http://localhost:8080/
```

REGISTERING A SERVLET WITH THE HTTPSERVICE MANUALLY

See Apache Felix HTTP Service.

USING THE PAX WEB WHITEBOARD EXTENDER

The Pax Web whiteboard extender is part of the war feature. So use the following command to install:

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

The Pax Web whiteboard extender listens to services
of interface type HttpServlet and Filter.
It will register each of these interfaces with the HttpService and remove them as soon as the service goes down.
So it is much more convenient than registering with the HttpService directly.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <service interface="javax.servlet.http.HttpServlet">
        <service-properties>
            <entry key="alias" value="/myservlet"/>
```

```
        </service-properties>
        <bean id="myServlet" class="com.example.MyServlet"/>
    </service>
</blueprint>
```

The above snippet publishes the Servlet MyServlet on http://localhost:8080/myServlet.

  Please keep in mind that the Whiteboard pattern for Servlets is not standardized and only works with Pax Web.

Web applications

INSTALLING WAR SUPPORT

The following steps will install the "war" feature (support for deploying WAR files with Servlet and JSPs into a Jetty server) into your Karaf instance.

1. List the available features -

```
karaf@root> features:list
 State        Name
. . .
[uninstalled] [2.2.0] obr         karaf-2.2.0
[uninstalled] [2.2.0] config      karaf-2.2.0
[uninstalled] [2.2.0] http        karaf-2.2.0
[uninstalled] [2.2.0] war         karaf-2.2.0
[uninstalled] [2.2.0] webconsole karaf-2.2.0
[installed  ] [2.2.0] ssh         karaf-2.2.0
. . .
```

2. Install the war feature (and the sub-features it requires) -

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

Note: you can use the -v or --verbose to see exactly what Karaf does

```
karaf@root> features:install -v war
Installing feature war 2.1.99-SNAPSHOT
Installing feature http 2.1.99-SNAPSHOT
Installing feature jetty 7.1.6.v20100715
Installing bundle mvn:org.apache.geronimo.specs/
geronimo-servlet_2.5_spec/1.1.2
Found installed bundle: org.apache.servicemix.bundles.asm
[10]
Installing bundle mvn:org.eclipse.jetty/jetty-util/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/jetty-io/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/jetty-http/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/
jetty-continuation/7.1.6.v20100715
```

```
Installing bundle mvn:org.eclipse.jetty/jetty-server/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/jetty-security/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/jetty-servlet/
7.1.6.v20100715
Installing bundle mvn:org.eclipse.jetty/jetty-xml/
7.1.6.v20100715
Checking configuration file mvn:org.apache.karaf/
apache-karaf/2.1.99-SNAPSHOT/xml/jettyconfig
Installing bundle mvn:org.ops4j.pax.web/pax-web-api/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-spi/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-runtime/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-jetty/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-jsp/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/
pax-web-extender-war/0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/
pax-web-extender-whiteboard/0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.web/pax-web-deployer/
0.8.2-SNAPSHOT
Installing bundle mvn:org.ops4j.pax.url/pax-url-war/1.2.4
```

3. Verify the features were installed

```
servicemix> features/list
 State          Name
. . .
[installed ] [2.2.0] http karaf-2.2.0
[installed ] [2.2.0] war  karaf-2.2.0
. . .
```

4. Verify the installed bundles were started

```
karaf@root> osgi:list
START LEVEL 100
   ID   State          Level  Name
. . .
[ 32] [Active ] [ ] [ 60] geronimo-servlet_2.5_spec
```

```
(1.1.2)
[ 33] [Active ] [ ] [ 60] Apache ServiceMix :: Bundles ::
jetty (6.1.22.2)
[ 34] [Active ] [ ] [ 60] OPS4J Pax Web - API (1.0.0)
[ 35] [Active ] [ ] [ 60] OPS4J Pax Web - Service SPI
(1.0.0)
[ 36] [Active ] [ ] [ 60] OPS4J Pax Web - Runtime (1.0.0)
[ 37] [Active ] [ ] [ 60] OPS4J Pax Web - Jetty (1.0.0)
[ 38] [Active ] [ ] [ 60] OPS4J Pax Web - Jsp Support
(1.0.0)
[ 39] [Active ] [ ] [ 60] OPS4J Pax Web - Extender - WAR
(1.0.0)
[ 40] [Active ] [ ] [ 60] OPS4J Pax Web - Extender -
Whiteboard (1.0.0)
[ 42] [Active ] [ ] [ 60] OPS4J Pax Web - FileInstall
Deployer (1.0.0)
[ 41] [Active ] [ ] [ 60] OPS4J Pax Url - war:, war-i:
(1.2.4)
. . .
```

5. The Jetty server should now be listening on http://localhost:8181/, but with no published applications available.

```
HTTP ERROR: 404
NOT_FOUND
RequestURI=/
Powered by jetty://
```

DEPLOYING A WAR TO THE INSTALLED WEB FEATURE

The following steps will describe how to install a simple WAR file (with JSPs or Servlets) to the just installed web feature.

1. To deploy a WAR (JSP or Servlet) to Jetty, update its MANIFEST.MF to include the required OSGi headers as described here - http://team.ops4j.org/wiki/display/ops4j/Pax+Web+Extender+-+War+-+OSGi-fy
2. Copy the updated WAR (archive or extracted files) to the *deploy* directory.

If you want to deploy a sample web application into Karaf, you could use the following command:

```
karaf@root> osgi:install -s webbundle:http://tomcat.apache.org/
tomcat-5.5-doc/appdev/sample/
sample.war?Bundle-SymbolicName=tomcat-sample&Webapp-Context=/sample
```

Then open your web browser and point to `http://localhost:8181/sample/`
`index.html`.

JRE and system packages tuning

Some packages are part of the JRE, but could be provided by tier libraries.

Some examples of package in this case are:
- `javax.xml.stream` package is provided by the JRE, but also by Apache Geronimo Spec bundles (and the implementation of provided by Woodstox).
- `com.sun.org.apache.xerces.internal.dom` package is provided by the JRE, but also by Apache Xerces.
- `javax.activation` package is provided by the JRE, but also by Apache Geronimo Spec bundles.
- `javax.annotation` package is provided by the JRE, but also by Apache Geronimo Spec bundles.
- etc ...

We should define in the Karaf instance which packages should be provided by the JRE or by tier bundles.

Moreover, we should manage the exported packages of JRE depending of the JRE version: the JRE 1.6 provides more packages
than the 1.5 one.

To customize this in Karaf, you can tune the `etc/jre.properties` file.

In the `etc/jre.properties` file, you have two properties `jre-1.5` and `jre-1.6`.

They allow you to define the packages exported by the JRE (in function of the runtime version).

Each property provide the exhaustive list of packages provided by the JRE (and so part of the sytem packages).

For instance, a "standard" Karaf distribution provides the following `etc/jre.properties`:

```
#
# Java platform package export properties.
#

# Standard package set.  Note that:
#   - javax.transaction* is exported with a mandatory attribute
jre-1.5= \
 javax.accessibility, \
 javax.activity, \
 javax.crypto, \
 javax.crypto.interfaces, \
 javax.crypto.spec, \
```

```
javax.imageio, \
javax.imageio.event, \
javax.imageio.metadata, \
javax.imageio.plugins.bmp, \
javax.imageio.plugins.jpeg, \
javax.imageio.spi, \
javax.imageio.stream, \
javax.management, \
javax.management.loading, \
javax.management.modelmbean, \
javax.management.monitor, \
javax.management.openmbean, \
javax.management.relation, \
javax.management.remote, \
javax.management.remote.rmi, \
javax.management.timer, \
javax.naming, \
javax.naming.directory, \
javax.naming.event, \
javax.naming.ldap, \
javax.naming.spi, \
javax.net, \
javax.net.ssl, \
javax.print, \
javax.print.attribute, \
javax.print.attribute.standard, \
javax.print.event, \
javax.rmi, \
javax.rmi.CORBA, \
javax.rmi.ssl, \
javax.security.auth, \
javax.security.auth.callback, \
javax.security.auth.kerberos, \
javax.security.auth.login, \
javax.security.auth.spi, \
javax.security.auth.x500, \
javax.security.cert, \
javax.security.sasl, \
javax.sound.midi, \
javax.sound.midi.spi, \
javax.sound.sampled, \
javax.sound.sampled.spi, \
javax.sql, \
```

```
javax.sql.rowset, \
javax.sql.rowset.serial, \
javax.sql.rowset.spi, \
javax.swing, \
javax.swing.border, \
javax.swing.colorchooser, \
javax.swing.event, \
javax.swing.filechooser, \
javax.swing.plaf, \
javax.swing.plaf.basic, \
javax.swing.plaf.metal, \
javax.swing.plaf.multi, \
javax.swing.plaf.synth, \
javax.swing.table, \
javax.swing.text, \
javax.swing.text.html, \
javax.swing.text.html.parser, \
javax.swing.text.rtf, \
javax.swing.tree, \
javax.swing.undo, \
javax.transaction; javax.transaction.xa; partial=true;
mandatory:=partial, \
javax.xml, \
javax.xml.datatype, \
javax.xml.namespace, \
javax.xml.parsers, \
javax.xml.transform, \
javax.xml.transform.dom, \
javax.xml.transform.sax, \
javax.xml.transform.stream, \
javax.xml.validation, \
javax.xml.xpath, \
org.ietf.jgss, \
org.omg.CORBA, \
org.omg.CORBA_2_3, \
org.omg.CORBA_2_3.portable, \
org.omg.CORBA.DynAnyPackage, \
org.omg.CORBA.ORBPackage, \
org.omg.CORBA.portable, \
org.omg.CORBA.TypeCodePackage, \
org.omg.CosNaming, \
org.omg.CosNaming.NamingContextExtPackage, \
org.omg.CosNaming.NamingContextPackage, \
```

```
 org.omg.Dynamic, \
 org.omg.DynamicAny, \
 org.omg.DynamicAny.DynAnyFactoryPackage, \
 org.omg.DynamicAny.DynAnyPackage, \
 org.omg.IOP, \
 org.omg.IOP.CodecFactoryPackage, \
 org.omg.IOP.CodecPackage, \
 org.omg.Messaging, \
 org.omg.PortableInterceptor, \
 org.omg.PortableInterceptor.ORBInitInfoPackage, \
 org.omg.PortableServer, \
 org.omg.PortableServer.CurrentPackage, \
 org.omg.PortableServer.POAManagerPackage, \
 org.omg.PortableServer.POAPackage, \
 org.omg.PortableServer.portable, \
 org.omg.PortableServer.ServantLocatorPackage, \
 org.omg.SendingContext, \
 org.omg.stub.java.rmi, \
 org.omg.stub.javax.management.remote.rmi, \
 org.w3c.dom, \
 org.w3c.dom.bootstrap, \
 org.w3c.dom.css, \
 org.w3c.dom.events, \
 org.w3c.dom.html, \
 org.w3c.dom.ls, \
 org.w3c.dom.ranges, \
 org.w3c.dom.stylesheets, \
 org.w3c.dom.traversal, \
 org.w3c.dom.views, \
 org.xml.sax, \
 org.xml.sax.ext, \
 org.xml.sax.helpers

# Standard package set.  Note that:
#   - javax.transaction* is exported with a mandatory attribute
jre-1.6= \
 javax.accessibility, \
 javax.activation, \
 javax.activity, \
 javax.annotation, \
 javax.annotation.processing, \
 javax.crypto, \
 javax.crypto.interfaces, \
```

```
javax.crypto.spec, \
javax.imageio, \
javax.imageio.event, \
javax.imageio.metadata, \
javax.imageio.plugins.bmp, \
javax.imageio.plugins.jpeg, \
javax.imageio.spi, \
javax.imageio.stream, \
javax.jws, \
javax.jws.soap, \
javax.lang.model, \
javax.lang.model.element, \
javax.lang.model.type, \
javax.lang.model.util, \
javax.management, \
javax.management.loading, \
javax.management.modelmbean, \
javax.management.monitor, \
javax.management.openmbean, \
javax.management.relation, \
javax.management.remote, \
javax.management.remote.rmi, \
javax.management.timer, \
javax.naming, \
javax.naming.directory, \
javax.naming.event, \
javax.naming.ldap, \
javax.naming.spi, \
javax.net, \
javax.net.ssl, \
javax.print, \
javax.print.attribute, \
javax.print.attribute.standard, \
javax.print.event, \
javax.rmi, \
javax.rmi.CORBA, \
javax.rmi.ssl, \
javax.script, \
javax.security.auth, \
javax.security.auth.callback, \
javax.security.auth.kerberos, \
javax.security.auth.login, \
javax.security.auth.spi, \
```

```
 javax.security.auth.x500, \
 javax.security.cert, \
 javax.security.sasl, \
 javax.sound.midi, \
 javax.sound.midi.spi, \
 javax.sound.sampled, \
 javax.sound.sampled.spi, \
 javax.sql, \
 javax.sql.rowset, \
 javax.sql.rowset.serial, \
 javax.sql.rowset.spi, \
 javax.swing, \
 javax.swing.border, \
 javax.swing.colorchooser, \
 javax.swing.event, \
 javax.swing.filechooser, \
 javax.swing.plaf, \
 javax.swing.plaf.basic, \
 javax.swing.plaf.metal, \
 javax.swing.plaf.multi, \
 javax.swing.plaf.synth, \
 javax.swing.table, \
 javax.swing.text, \
 javax.swing.text.html, \
 javax.swing.text.html.parser, \
 javax.swing.text.rtf, \
 javax.swing.tree, \
 javax.swing.undo, \
 javax.tools, \
 javax.transaction; javax.transaction.xa; partial=true;
mandatory:=partial, \
 javax.xml, \
 javax.xml.bind, \
 javax.xml.bind.annotation, \
 javax.xml.bind.annotation.adapters, \
 javax.xml.bind.attachment, \
 javax.xml.bind.helpers, \
 javax.xml.bind.util, \
 javax.xml.crypto, \
 javax.xml.crypto.dom, \
 javax.xml.crypto.dsig, \
 javax.xml.crypto.dsig.dom, \
 javax.xml.crypto.dsig.keyinfo, \
```

```
javax.xml.crypto.dsig.spec, \
javax.xml.datatype, \
javax.xml.namespace, \
javax.xml.parsers, \
javax.xml.soap, \
javax.xml.stream, \
javax.xml.stream.events, \
javax.xml.stream.util, \
javax.xml.transform, \
javax.xml.transform.dom, \
javax.xml.transform.sax, \
javax.xml.transform.stax, \
javax.xml.transform.stream, \
javax.xml.validation, \
javax.xml.ws, \
javax.xml.ws.handler, \
javax.xml.ws.handler.soap, \
javax.xml.ws.http, \
javax.xml.ws.soap, \
javax.xml.ws.spi, \
javax.xml.ws.wsaddressing, \
javax.xml.xpath, \
org.ietf.jgss, \
org.omg.CORBA, \
org.omg.CORBA_2_3, \
org.omg.CORBA_2_3.portable, \
org.omg.CORBA.DynAnyPackage, \
org.omg.CORBA.ORBPackage, \
org.omg.CORBA.portable, \
org.omg.CORBA.TypeCodePackage, \
org.omg.CosNaming, \
org.omg.CosNaming.NamingContextExtPackage, \
org.omg.CosNaming.NamingContextPackage, \
org.omg.Dynamic, \
org.omg.DynamicAny, \
org.omg.DynamicAny.DynAnyFactoryPackage, \
org.omg.DynamicAny.DynAnyPackage, \
org.omg.IOP, \
org.omg.IOP.CodecFactoryPackage, \
org.omg.IOP.CodecPackage, \
org.omg.Messaging, \
org.omg.PortableInterceptor, \
org.omg.PortableInterceptor.ORBInitInfoPackage, \
```

```
org.omg.PortableServer, \
org.omg.PortableServer.CurrentPackage, \
org.omg.PortableServer.POAManagerPackage, \
org.omg.PortableServer.POAPackage, \
org.omg.PortableServer.portable, \
org.omg.PortableServer.ServantLocatorPackage, \
org.omg.SendingContext, \
org.omg.stub.java.rmi, \
org.omg.stub.javax.management.remote.rmi, \
org.w3c.dom, \
org.w3c.dom.bootstrap, \
org.w3c.dom.css, \
org.w3c.dom.events, \
org.w3c.dom.html, \
org.w3c.dom.ls, \
org.w3c.dom.ranges, \
org.w3c.dom.stylesheets, \
org.w3c.dom.traversal, \
org.w3c.dom.views, \
org.w3c.dom.xpath, \
org.xml.sax, \
org.xml.sax.ext, \
org.xml.sax.helpers
```

For instance, if we deploy the Apache Geronimo Specs bundle providing the
`javax.xml.stream` package, we have to comment
the `javax.xml.stream` package in the the jre-1.6 definition. Indeed, jre-1.5 doesn't
provide `javax.xml.stream`, it's
new in the JRE 1.6:

```
jre-1.6= \
[...]
# javax.xml.stream, \
# javax.xml.stream.events, \
# javax.xml.stream.util, \
[...]
```

Monitoring and Administration using JMX

Apache Karaf provides a large set of MBeans that allow you to fully monitore and administrate Karaf using any JMX client
(like jconsole for instance).

You can find more or less the same actions that you can do using the shell commands on the JMX layer.

Apache Karaf provides the following MBeans:

- org.apache.karaf:type=admin to administrate the child instances
- org.apache.karaf:type=bundles to manipulate the OSGi bundles
- org.apache.karaf:type=config to manipulate the Karaf configuration files (in the etc folder) and the ConfigAdmin layer
- org.apache.karaf:type=dev to get information and manipulate the OSGi framework
- org.apache.karaf:type=diagnostic to create information file (dump) about Karaf activity
- org.apache.karaf:type=features to manipulate the Karaf features
- org.apache.karaf:type=log to manipulate to logging layer
- org.apache.karaf:type=packages to manipulate to PackageAdmin layer and get information about exported and imported packages
- org.apache.karaf:type=services to get information about the OSGi services
- org.apache.karaf:type=system to shutdown the Karaf container itself
- org.apache.karaf:type=web to get information about the Web bundles (installed with the war feature)
- org.apache.karaf:type=obr to manipulate the OBR layer (installed with the obr feature)

Developers Guide

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 \
  -DartifactId=karaf-command-archetype \
  -DarchetypeVersion=2.3.12 \
  -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:2.3.12]
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:2.3.12
[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 /home/jbonofre/
workspace/karaf/test/com.mycompany.command/pom.xml
[INFO] project created from Archetype in dir:
/home/jbonofre/workspace/karaf/test/
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=2.3.12 \
    -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:3.0.0-SNAPSHOT]
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:3.0.0-SNAPSHOT
[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: /home/jbonofre/
workspace/karaf/test/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=2.3.12 \
    -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:3.0.0-SNAPSHOT]
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:3.0.0-SNAPSHOT
[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: /home/jbonofre/
workspace/karaf/test/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=2.3.12 \
```

```
    -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:3.0.0-SNAPSHOT]
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:3.0.0-SNAPSHOT
[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: /home/jbonofre/
workspace/karaf/test/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=2.3.12 \
    -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:3.0.0-SNAPSHOT]
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:3.0.0-SNAPSHOT
[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: /home/jbonofre/
workspace/karaf/test/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]
------------------------------------------------------------------------
```

Branding the Console

This chapter will show you how to customize the user interface of the Karaf console, making
changes such as a new welcome message and console prompt. This is what we refer to as "branding" Karaf.

CREATE YOUR BRANDING BUNDLE

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

So you need to create a very simple bundle containing just a `org/apache/karaf/branding/branding.properties` file.

The Maven POM of your branding bundle should look like this:

```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>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>2.2</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>
```

Now, add a `src/main/resources/org/apache/karaf/branding/`
`branding.properties` 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 (2.3.12)\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>
  {

}
```

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

- welcome is the welcome message displayed when you start Karaf.
- 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 to bold and \u001B[0m
switches it back to normal.

Some examples of customized prompts:

```
# 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>
```

CONFIGURING KARAF TO USE THE BRANDING BUNDLE

In order for Karaf to pick up the branding jar please edit the $KARAF_HOME/etc/custom.properties file to include the following:
    org.osgi.framework.system.packages.extra = \
org.apache.karaf.branding

INSTALLING THE BRANDING BUNDLE

Build your branding bundle:

```
mvn install
```

and simply drop the generated jar file into the Karaf lib directory.
    Start Karaf and you will see your branded Karaf console.

Extending the console

This chapter will guide you through the steps needed to extend the console and create a new shell.

We will leverage Maven, Blueprint and OSGi, so you will need some knowledge of those products.

You may also find some information about the console at http://felix.apache.org/site/rfc-147-overview.html.

CREATE THE PROJECT USING MAVEN

We first need to create a project using Maven. Let's leverage Maven archetypes for that.

Command line

Using the command line, we can create our project:

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

This generate the main `pom.xml` and some additional packages.

Interactive shell

You can also use the interactive mode for creating the skeleton project:

```
mvn archetype:generate
```

Use the following values when prompted:

```
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
```

Manual creation

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

```xml
<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>2.3.12</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>2.4.0</version>
        <configuration>
          <instructions>
            <Import-Package>
              org.apache.felix.service.command,
              org.apache.felix.gogo.commands,
              org.apache.karaf.shell.console,
              *
            </Import-Package>
```

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

</project>
```

CONFIGURING FOR JAVA 5

We are using annotations to define commands, so we need to ensure Maven will actually use JDK 1.5 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>1.5</target>
        <source>1.5</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`

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

import org.apache.felix.gogo.commands.Command;
import org.apache.karaf.shell.console.OsgiCommandSupport;

@Command(scope = "test", name = "hello", description="Says hello")
public class HelloShellCommand extends OsgiCommandSupport {

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

CREATING THE ASSOCIATED BLUEPRINT CONFIGURATION FILES

The blueprint configuration file will be used to create the command and register it in the OSGi registry, which is the way to make the command available to Karaf console. This blueprint file must be located in the `OSGI-INF/blueprint/` directory inside the bundle.

If you don't have the `src/main/resources` directory yet, create it.

```
mkdir src/main/resources
```

Then, re-generate the IDE project files and reload it so that this folder is now recognized as a source folder.

Inside this directory, create the `OSGI-INF/blueprint/` directory and put the following file inside (the name of this file has no impact at all):

```xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/
v1.0.0">
        <command name="test/hello">
            <action
class="org.apache.karaf.shell.samples.HelloShellCommand"/>
        </command>
    </command-bundle>
```

```
</blueprint>
```

COMPILING THE JAR

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 IN KARAF

Launch a Karaf instance and run the following command to install the newly created bundle:

```
karaf@root> osgi: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.felix.gogo.commands.Command;
import org.apache.felix.gogo.commands.Argument;
import org.apache.karaf.shell.console.OsgiCommandSupport;

@Command(scope = "test", name = "hello", description="Says
hello")
public class HelloShellCommand extends OsgiCommandSupport {

    @Argument(index = 0, name = "arg", description = "The
command argument", required = false, multiValued = false)
    String arg = null;

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

The Blueprint configuration file is the same as previously.

COMPLETER BEAN

A completer is a bean which implements the Completer interface:

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

```java
import org.apache.karaf.shell.console.completer.StringsCompleter;
import org.apache.karaf.shell.console.Completer;

/**
 * <p>
 * A very simple completer.
 * </p>
 */
public class SimpleCompleter implements Completer {

  /**
   * @param buffer the beginning string typed by the user
   * @param cursor the position of the cursor
   * @param candidates the list of completions proposed to the
user
   */
 public int complete(String buffer, int cursor, List candidates)
{
    StringsCompleter delegate = new StringsCompleter();
    delegate.getStrings().add("one");
    delegate.getStrings().add("two");
    delegate.getStrings().add("three");
    return delegate.complete(buffer, cursor, candidates);
 }

}
```

BLUEPRINT CONFIGURATION FILE

Using Blueprint, you can "inject" the completer linked to your command. The same completer could be used for several commands and a command can have several completers:

```xml
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <command-bundle xmlns="http://karaf.apache.org/xmlns/shell/
v1.0.0">
        <command name="test/hello">
            <action
class="org.apache.karaf.shell.samples.HelloShellCommand"/>
            <completers>
                <ref component-id="simpleCompleter"/>
                <null/>
```

```
            </completers>
        </command>
    </command-bundle>

    <bean id="simpleCompleter"
class="org.apache.karaf.shell.samples.SimpleCompleter"/>

</blueprint>
```

TEST IN KARAF

Launch a Karaf instance and run the following command to install the newly created bundle:

```
karaf@root> osgi: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
```

Custom distributions

As Karaf is an OSGi container, it's heavily used as an 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

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>
```

```xml
        <karaf.version>2.2.2</karaf.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.karaf</groupId>
            <artifactId>apache-karaf</artifactId>
            <version>2.3.12</version>
            <type>tar.gz</type>
        </dependency>
        <dependency>
            <groupId>org.apache.karaf.assemblies.features</groupId>
            <artifactId>standard</artifactId>
            <version>2.3.12</version>
            <type>xml</type>
            <classifier>features</classifier>
        </dependency>
    </dependencies>

    <build>
        <resources>
            <resource>
                <directory>/home/jbonofre/Workspace/karaf/target/
checkout/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>
```

```xml
      </plugin>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.3.12</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.assemblies.features/standard/
2.3.12/xml/features</descriptor>
                <descriptor>file:/home/jbonofre/Workspace/karaf/
target/checkout/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>
```

```xml
                    <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/
2.3.12/xml/features,mvn:my.groupId/my-features/2.3.12/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 scripts, 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>
```

```xml
            <format>tar.gz</format>
    </formats>

    <fileSets>

        <!-- Expanded Karaf Standard Distribution -->
        <fileSet>
            <directory>target/dependencies/
apache-karaf-2.3.12</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-2.3.12</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/target/
checkout/manual/target/dependencies/apache-karaf-2.3.12/bin/
karaf</source>
            <outputDirectory>/bin/</outputDirectory>
            <destName>my-custom</destName>
            <fileMode>0755</fileMode>
            <lineEnding>unix</lineEnding>
        </file>
        <file>
            <source>/home/jbonofre/Workspace/karaf/target/
checkout/manual/target/classes/features.xml</source>
            <outputDirectory>/system/my.groupid/my-features/
2.3.12</outputDirectory>
            <destName>my-features-2.3.12-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

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:

You can find the schema at the following location.

Here are two examples using this schema:

```xml
<?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">

    <type-converters>
```

```
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
    </type-converters>

    <!-- 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>

    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory"/>
    </service>

</blueprint>
```

NB: Don't forge to expose the BackingEngine as an OSGi service.

```
<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
<?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="]"/>

    <type-converters>
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
    </type-converters>

    <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
BackingEngineFactory	org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory

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]...
```

```xml
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                flags="required">
        users = $[karaf.base]/etc/users.properties
    </jaas:module>
</jaas:config>
```

Note for windows users: Do not forget the typeconverter from the first config snippet. It makes sure that windows paths are handled correctly.

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
<?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
BackingEngineFactory	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
```

```
className="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
BackingEngineFactory	org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory

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
BackingEngineFactory	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 optional 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 will 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:

```xml
<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).

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
BackingEngineFactory	org.apache.karaf.jaas.modules.syncope.SyncopeBackingEngineFactory

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 fulfills simple needs. However, in some cases, you may need 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:

```xml
<jaas:config name="karaf">
    <jaas:module
className="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:

```java
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())));
        }
}
```

<u>Grouped Roles</u>

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());
        }
}
```

Using the features-maven-plugin

The `features-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
`features:add-features-to-repo`	Copies all the bundles required for a given set of features into a directory (e.g. for creating your own Karaf-based distribution)
`features:generate-features-file`	Deprecated - use `features:generate-features-xml` instead
`features:generate-features-xml`	Generates a features XML descriptor for a set of bundles
`features:validate`	Validate a features XML descriptor by checking if all the required imports can be matched to exports
{{	features:create-kar

CONFIGURE THE FEATURES-MAVEN-PLUGIN

In order to use the `features-maven-plugin`, you have to define the plugin in your project's `pom.xml` file:

```xml
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.3.12</version>

        <executions>
          <!-- add execution definitions here -->
        </executions>
      </plugin>
    </plugins>
```

```
    </build>
</project>
```

GOAL `FEATURES:ADD-FEATURES-TO-REPO`

The `features:add-features-to-repo` 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.

```xml
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.3.12</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:my.groupid/my.artifactid/1.0.0/xml/
features</descriptor>
              </descriptors>
              <features>
                <feature>spring</feature>
                <feature>war</feature>
                <feature>my</feature>
                <feature>other/1.0-SNAPSHOT</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 NB: Karaf core features descriptors (standard and enterprise) are automatically added in this list
features	String[]	List of features that bundles should be copied to the repository directory. A feature could be just a feature name or a name/version. If it's just a name, the features-maven-plugin will take the first feature with the given name, whatever the version is.
repository	File	The directory where the bundles will be copied by the plugin goal
karafVersion	String	Target Karaf version to use to resolve the Karaf core features descriptors (standard and enterprise)

GOAL `FEATURES:GENERATE`

The `features:generate-features-xml` goal generates a features XML file for every bundle listed in the project's dependencies. In order to satisfy the required imports in these bundles, the plugin will add bundles:

- bundles provided by Apache Karaf
- a explicit list of bundles
- bundles discovered in the POM's transitive dependencies

Afterwards, the generated file will be attached to the build as an additional build artifact (by default as `group:artifact:version:xml:features`).

Example

The example below generates one feature that installs bundle `mvn:org.apache:bundle1:1.0` in a features XML file called `target/features.xml`. It will find bundle in Apache Karaf 2.3.12, the transitive dependencies for this POM and the bundles listed in `src/main/resources/bundles.properties`.

```xml
<project>
  <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>features-maven-plugin</artifactId>
        <version>2.3.12</version>
        <executions>
          <execution>
            <id>generate</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>generate-features-xml</goal>
            </goals>
            <configuration>
              <bundles>src/main/resources/
bundles.properties</bundles>
              <kernelVersion>2.3.12</kernelVersion>
              <outputFile>target/features.xml</outputFile>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description

outputFile	File	Name of the features XML file that is being generated Default value: `/home/jbonofre/Workspace/karaf/target/checkout/manual/target/classes/feature.xml`
attachmentArtifactType	String	The artifact type for attaching the generated file to the project Default value: {{xml}}
attachmentArtifactClassifier	String	The artifact classifier for attaching the generated file to the project Default value: `features`
kernelVersion	String	The version of Karaf that is used to determine system bundles and default provided features
bundles	File	A properties file that contains a list of bundles that will be used to generate the features.xml file

GOAL `FEATURES:VALIDATE`

The `features:validate` 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 standard features (standard and enterprise) in the repositories set.
It means that it's not necessary 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.

```xml
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.3.12</version>
          <executions>
            <execution>
              <id>validate</id>
              <phase>process-resources</phase>
              <goals>
                <goal>validate</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/target/checkout/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 value is the version of the plugin
`repositories`	`String[]`	Additional features XML descriptors that will be used during the validation process

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 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
{noformat
or on Windows
```

> bin\karaf.bat debug
{noformat

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. Nevertheless, 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 be able to 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'
```

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:

Programmatically connect to the console

A connection to the Karaf console can also be done programmatically.
The following code is a simplified version of the code from the client library.

```java
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);
    }

}
```

You can find a more complete example at the following location.

Writing integration tests

We recommend using PAX Exam to write integration tests when developing applications using Karaf.

Starting with Karaf 2.3.0 and 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:

```xml
<!-- Karaf Test Framework Version -->
<dependency>
  <groupId>org.apache.karaf.tooling.exam</groupId>
  <artifactId>org.apache.karaf.tooling.exam.container</artifactId>
  <version>2.3.12</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>2.6.0</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:

```xml
<dependency>
  <groupId>org.apache.karaf</groupId>
  <artifactId>apache-karaf</artifactId>
  <version>2.3.12</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>${plugin.depends.version}</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/
2.3.12/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/
2.3.12/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:

```java
@Configuration
public Option[] config() {
    return new Option[]{
        new
KarafDistributionKitConfigurationOption("mvn:org.apache.karaf/
apache-karaf/2.3.12/zip",

Platform.WINDOWS).executable("bin\\karaf.bat").filesToMakeExecutable("bin\\admin.
        new
KarafDistributionKitConfigurationOption("mvn:org.apache.karaf/
apache-karaf/2.3.12/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:

```java
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/
2.3.12/zip"),
        configureConsole().ignoreLocalConsole().startRemoteShell()
};
}
```

```
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
2.3.12/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/
2.3.12/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/
2.3.12/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/
2.3.12/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:

```java
@Configuration
public Option[] config() {
    return new Option[]{
karafDistributionConfiguration("mvn:org.apache.karaf/apache-karaf/
2.3.12/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.

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).

For instance, you can define a bundle like this (using Apache Felix maven-bundle-plugin):

```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.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/target/
checkout/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>2.2.0</version>
```

```
                <extensions>true</extensions>
                <configuration>
                    <instructions>

<Bundle-SymbolicName>manual</Bundle-SymbolicName>
                          ...
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

And simply add a src/main/resources/OSGI-INF/bundle.info file containing, for instance:

```
\u001B[1mSYNOPSIS\u001B[0m
    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.

\u001B[1mDESCRIPTION\u001B[0m
    Long description of your bundle, including usage, etc.

\u001B[1mSEE ALSO\u001B[0m
    \u001B[36mhttp://yourside\u001B[0m
    \u001B[36mhttp://yourside/docs\u001B[0m
```

You can display this extended information using:

```
root@karaf> osgi:info
```

Creating bundles for third party dependencies

Karaf supports the wrap: protocol execution.

It allows for directly deploying third party dependencies, like Apache Commons Lang:

```
root@karaf> osgi:install wrap:mvn:commons-lang/commons-lang/2.4
```

You can specify OSGi statements in the wrap URL:

from the shell

```
root@karaf> osgi:install -s '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>
```

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, you can 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-Depen
                </instructions>
                <unpackBundle>true</unpackBundle>
            </configuration>
        </plugin>
    </build>

</project>
```

You have now a OSGi bundle for commons-lang that you can deploy directly:

```
root@karaf> osgi: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.

Shell syntax

EASY TO USE INTERACTIVELY - NO UNNECESSARY SYNTAX

```
// simple command
karaf@root> echo hello world
hello world

// session variables
karaf@root> msg = "hello world"
hello world
karaf@root> echo $msg
hello world

// execution quotes () - similar to bash backquotes
karaf@root> (bundle 1) location
mvn:org.ops4j.pax.url/pax-url-mvn/1.1.3
```

LIST, MAPS, PIPES AND CLOSURES

```
// lists - []
karaf@root> list = [1 2 a b]
1
2
a
b

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

// pipes
karaf@root> bundles | grep felix
000000 ACT org.apache.felix.framework-3.0.2
000005 ACT org.apache.felix.configadmin-1.2.4
000006 ACT org.apache.felix.fileinstall-3.0.2

// closures - {}
```

```
karaf@root> echo2 = { echo xxx $args yyy }
org.apache.felix.gogo.runtime.shell.Closure@2ffb36c2
karaf@root> echo2 hello world
xxx hello world yyy
```

LEVERAGES EXISTING JAVA CAPABILITIES, VIA REFLECTION

```
// exception handling - console shows summary, but full context
available
karaf@root> start xxx
Error executing command osgi:start: unable to convert argument
ids with value '[xxx]' to type java.util.List<java.lang.Long>
karaf@root> $karaf.lastException printStackTrace
org.apache.felix.gogo.commands.CommandException: Unable to
convert argument ids with value '[xxx]' to type
java.util.List<java.lang.Long>
        at
org.apache.felix.gogo.commands.basic.DefaultActionPreparator.prepare(Defaul
        at
org.apache.felix.gogo.commands.basic.AbstractCommand.execute(AbstractComman
        at
org.apache.felix.gogo.runtime.shell.CommandProxy.execute(CommandProxy.java:
        at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:229)
        at
org.apache.felix.gogo.runtime.shell.Closure.executeStatement(Closure.java:1
        at
org.apache.felix.gogo.runtime.shell.Pipe.run(Pipe.java:101)
        at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:79)
        at
org.apache.felix.gogo.runtime.shell.CommandSessionImpl.execute(CommandSessi
        at
org.apache.karaf.shell.console.jline.Console.run(Console.java:169)
        at java.lang.Thread.run(Thread.java:637)
Caused by: java.lang.Exception: Unable to convert from [xxx] to
java.util.List<java.lang.Long>(error converting collection entry)
        at
org.apache.aries.blueprint.container.AggregateConverter.convertToCollection
        at
org.apache.aries.blueprint.container.AggregateConverter.convert(AggregateCo
        at
```

```
org.apache.karaf.shell.console.commands.BlueprintCommand$BlueprintActionPre
        at
org.apache.felix.gogo.commands.basic.DefaultActionPreparator.prepare(Defaul
        ... 9 more
Caused by: java.lang.NumberFormatException: For input string:
"xxx"
        at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:4
        at java.lang.Long.parseLong(Long.java:410)
        at java.lang.Long.valueOf(Long.java:525)
        at
org.apache.aries.blueprint.container.AggregateConverter.convertFromString(A
        at
org.apache.aries.blueprint.container.AggregateConverter.convert(AggregateCo
        at
org.apache.aries.blueprint.container.AggregateConverter.convertToCollection
        ... 12 more

// add all public methods on java.lang.System as commands:
karaf@root> addcommand system (loadClass java.lang.System)
karaf@root> system:getproperty karaf.name
root

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