Apache Karaf
Version 2.2.0-fuse-00-61

Apache Karaf
Users' Guide

Copyright 2011 The Apache Software Foundation

Table of contents

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 `[home]/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, the 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 `[home]/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, still you will 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

Depending on the platform you plan to install and run Karaf you will select the appropriate installation image. 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 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 informations 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 remainder of this guide how to use and leverage Apache Karaf, we will just use the pre-built packaging for now.

In the console, run the following commands:

```
features:addurl mvn:org.apache.camel/camel-example-osgi/2.7.0/
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:

```
osgi:shutdown
```

SUMMARY

This document showed you 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 in less than five minutes.

Users Guide

Installation

This chapter describes how to install Apache Karaf for both Unix and Windows' platforms.
Here you will find information about what are pre requisite software, where to download Karaf from and how to install it.

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 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 SVN checkout, 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/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/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.

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 `[karaf_install_dir]/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:

```
osgi:shutdown
```

or simply:

```
shutdown
```

The shutdown command ask 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:

```
osgi: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:

```
osgi:shutdown 10:35
```

The following command will shutdown Karaf in 10 minutes:

```
osgi: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
```

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

Using the console

VIEWVING AVAILABLE COMMANDS

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

```
root@root> <tab>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 completion 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, completion may be available on options and arguments too.

GETTING HELP ON 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.

Enabling Colorized Console on Windows

The default Karaf installation does not produce colorized console output on Windows like it does on Unix based systems. To enable it, you must install LGPL licensed library JNA. This can be done using a few simple commands in the Karaf console:

You first need to install the JNA library:

```
osgi:install wrap:mvn:http://download.java.net/maven/
2!net.java.dev.jna/jna/3.1.0
```

Next you need either restart karaf or you run the following Karaf commands to refresh the Karaf Console:

```
osgi:list | grep "Apache Karaf :: Shell Console"
```

Take note of the ID of the bundle, in my case it was 14 and then run:

```
osgi:refresh 14
```

TODO: refactor that using a nicer script to find the correct bundle

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 you want):

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

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 conolse 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 `[karaf_install_dir]/etc/org.apache.karaf.shell.cfg` configuration file with the following defaults 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 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 to change 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 from the karaf-install-dir directory:

```
bin/client
```

More usually, 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 `karaf-install-dir/lib` directory:

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

Deployer

The following picture describes the architecture of the deployer.

SPRING DEPLOYER

Karaf includes a deployer that is able to deploy plain blueprint or spring-dm 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]
Spring-Context:
*;publish-context:=false;create-asynchronously:=true
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 spring configuration:

```
<spring:beans ...>
  <manifest>
    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.

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.

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

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

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.

```
karaf@root>admin:create finn
Creating new instance on SSH port 8106 and RMI port 1100 at:
/home/fuse/esb4/instances/finn
Creating dir:  /home/fuse/esb4/instances/finn/bin
Creating dir:  /home/fuse/esb4/instances/finn/etc
Creating dir:  /home/fuse/esb4/instances/finn/system
Creating dir:  /home/fuse/esb4/instances/finn/deploy
Creating dir:  /home/fuse/esb4/instances/finn/data
Creating file: /home/fuse/esb4/instances/finn/etc/
config.properties
Creating file: /home/fuse/esb4/instances/finn/etc/
java.util.logging.properties
Creating file: /home/fuse/esb4/instances/finn/etc/
org.apache.felix.fileinstall-deploy.cfg
```

```
Creating file: /home/fuse/esb4/instances/finn/etc/
org.apache.karaf.log.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/
org.apache.karaf.features.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/
org.ops4j.pax.logging.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/
org.ops4j.pax.url.mvn.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/
startup.properties
Creating file: /home/fuse/esb4/instances/finn/etc/
system.properties
Creating file: /home/fuse/esb4/instances/finn/etc/
org.apache.karaf.shell.cfg
Creating file: /home/fuse/esb4/instances/finn/etc/
org.apache.karaf.management.cfg
Creating file: /home/fuse/esb4/instances/finn/bin/karaf
Creating file: /home/fuse/esb4/instances/finn/bin/start
Creating file: /home/fuse/esb4/instances/finn/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] [-p password] 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 `karaf-install-dir/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 USERS AND PASSWORDS

The default security configuration uses a property file located at `karaf-install-dir/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 currenly 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 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 informations 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 require you to configure all your servers accordingly.

Karaf offers a simple way to configure additional security providers:

- put your provider jar in `karaf-install-dir/lib/ext`

- modify the `karaf-install-dir/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.felix.karaf.main.SimpleFileLock
karaf.lock.dir=<PathToLockFileDirectory>
karaf.lock.delay=10
```

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 only acquire the lock 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 `$KARAF_HOME/bin/karaf` script to have unique JMX remote port set if instances reside on the same host
- Update the `$KARAF_HOME/etc/system.properties` file as follows:

```
karaf.lock=true
karaf.lock.class=org.apache.felix.karaf.main.DefaultJDBCLock
```

```
karaf.lock.level=50
karaf.lock.delay=10
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:

- Will fail if JDBC driver is not on classpath.
- The database name "sample" will be created if it does not exist on the database.
- 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 manual restart.

JDBC locking on Oracle

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

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

```
karaf.lock=true
karaf.lock.class=org.apache.felix.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
```

As with the default JDBC locking setup, the Oracle JDBC driver JAR file must be in your classpath. You can ensure this by copying the `ojdbc14.jar` into Karaf's `lib` folder before starting Karaf.

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

TODO

MySQL

TODO

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=10
```

The `karaf.log.level` property tells the Karaf instance how far up the boot process to bring the OSGi container. Bundles assigned the same start level or lower will then also 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 as 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 be 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:set`: set the log levels
- `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
...
```

Now you can use the appender in your log4j config file like shown in the config examples above.

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 carefull 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 disable. To enable an URL for snapshots append @snapshots. For example

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

Repositories on the local 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=60`. 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 to define the start-level of a bundle is that you can deploy all your bundles including those of the project with the 'infrastructure' bundles required (e.g : camel, activemq)
at the same time and you will have the guaranty when you use Spring Dynamic Module (to register service through OSGI service layer), Blueprint that by example
Spring context will not be created without all the required services installed.

Bundle 'stop/start'

The OSGI specification allows to install 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>
```

Bundle 'dependency'

A bundle can be flagged as being a dependency. Such information can be used by resolvers to compute the final 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 will be 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 choosen. If nothing is specified, the highest available will be instaleld.

Configurations

The configuration section allows to deploy configuration for 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 than 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 informations about using the Configuration Admin service in Spring-DM, see the Spring-DM documentation.

Configuration files

Available since Karaf 2.2

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 help full 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.

```
<configfile finalname="/etc/jetty.xml">mvn:org.apache.karaf/
apache-karaf/${project.version}/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 again this URL.

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
```

Remark : 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>
```

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

SERVICE CONFIGURATION

A simple configuration file located in `[FELIX:karaf]/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 Apache Karaf distribution which includes pre-installed features. Such a process is detailed in the 6.2. Building custom distributions section.

XML Schema for provisioning

Following schema can be found in Karaf sources. It is also available public on url http://karaf.apache.org/xmlns/features/v1.0.0.

Developers Guide

Programmatically connect to the console

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

```
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.

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(Defau
        at
org.apache.felix.gogo.commands.basic.AbstractCommand.execute(AbstractComma
        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:
        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(CommandSess
        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.convertToCollectio
        at
```

```
org.apache.aries.blueprint.container.AggregateConverter.convert(AggregateC
        at
org.apache.karaf.shell.console.commands.BlueprintCommand$BlueprintActionPr
        at
org.apache.felix.gogo.commands.basic.DefaultActionPreparator.prepare(Defau
        ... 9 more
Caused by: java.lang.NumberFormatException: For input string:
"xxx"
        at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:
        at java.lang.Long.parseLong(Long.java:410)
        at java.lang.Long.valueOf(Long.java:525)
        at
org.apache.aries.blueprint.container.AggregateConverter.convertFromString(
        at
org.apache.aries.blueprint.container.AggregateConverter.convert(AggregateC
        at
org.apache.aries.blueprint.container.AggregateConverter.convertToCollectio
        ... 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
```

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 (to color, formatting, etc).

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

```
<?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>/mnt/hudson/workspace/
perfectus-esb-4.4.0-fuse/target/esb-4.4.0-fuse/karaf/manual/src/
main/resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
```

```
                <version>2.2.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>

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

</project>
```

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

```
\u001B[1mSYNOPSIS\u001B[0m
    This is a shared POM parent for FuseSource Maven projects

\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 support the wrap: protocol execution.

It allows you to directly deploy third party dependency, like Apache Commons Lang:

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

You can specify OSGi statements on the wrap URL:

```
root@karaf> osgi:install wrap:mvn:commons-lang/commons-lang/
2.4,Bundle-SymbolicName=commons-lang&Bundle-Version=2.4
```

Anyway, you can create a wrap bundle for a third party dependencies. This bundle is simply a Maven POM that shade an existing jar and package into a jar bundle.

For instance, to create an OSGi bundle to wrap 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 directory:

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

Some more infos 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.

Troubleshooting, Debugging,Profiling, and Monitoring

TROUBLESHOOTING

Logging

Logging is easy to control through the console, with commands grouped under *log* shell. To learn about the available logging commands type:

```
karaf@root> log<tab>

log:display                log:display-exception
log:get                    log:set
karaf@root>
```

Typical usage is:

```
   # Use log:set to dynamically change the global log level
# Execute the problematic operation
# Use log:display (or log:display-exception to display the log
```

Worst Case Scenario

If you end up with a Karaf in a really bad state (i.e. you can not boot it anymore) or you just want to revert to a clean state quickly, you can safely remove the `data` directory just in the installation directory. This folder contains transient data and will be recreated if you remove it and 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 we have to debug a project deployed top of Apache Karaf. Nervertheless, you will be blocked if you would like to debug the server Karaf. In this case, you can change the following parameter suspend=y in the karaf.bat script file. That will cause the JVM to pause just before running main() until you attach a debugger then it will resume the execution. This way you can set
your breakpoints anywhere in the code and you should hit them no matter how early in the startup they are

```
export DEFAULT_JAVA_DEBUG_OPTS='-Xdebug -Xnoagent
-Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005'
```

and on Windows,

```
set DEFAULT_JAVA_DEBUG_OPTS='-Xdebug -Xnoagent
-Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005'
```

PROFILING

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, uninstall Karaf features:

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 the 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:

```
<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/maven-v4_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>2.2.0-fuse-00-61</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>2.3.5</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`

```
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):

```
<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 allow 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;

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 it's the beginning string typed by the user
  * @param cursor it's 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:

```
<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>
        <completers>
          <ref component-id="simpleCompleter"/>
          <null/>
        </completers>
    </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
```

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

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:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.2.0-fuse-00-61</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.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.2.0-fuse-00-61</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>
              </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
`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` 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 list of bundles
- bundles discovered in the POM's transitive dependencies

Afterwards, the generated file is 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 2.2.0-fuse-00-61, the transitive dependencies for this POM and the bundles listed in `src/main/resources/bundles.properties`.

```
<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>2.2.0-fuse-00-61</version>
          <executions>
            <execution>
              <id>generate</id>
              <phase>generate-resources</phase>
              <goals>
                <goal>generate</goal>
              </goals>
              <configuration>
                <bundles>src/main/resources/
bundles.properties</bundles>
                <kernelVersion>2.2.0-fuse-00-61</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: /mnt/hudson/workspace/perfectus-esb-4.4.0-fuse/target/esb-4.4.0-fuse/karaf/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.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>features-maven-plugin</artifactId>
        <version>2.2.0-fuse-00-61</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: `/mnt/hudson/workspace/perfectus-esb-4.4.0-fuse/target/esb-4.4.0-fuse/karaf/manual/target/classes/features.xml`

`karafConfig`	`String`	The Karaf `config.properties` file to use during the validation process Default value: `config.properties`
`jreVersion`	`String`	The JRE version that is used during the validation process Default value: {{jre-1.5})
`karafVersion`	`String`	The target Karaf version used to get the Karaf core features (standard and enterprise) Default value is the version of the plugin
`repositories`	`String[]`	Additional features XML descriptors that will be used during the validation process

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 allow the deployment at runtime 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 example using this schema:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <!-- Bean to allow the ${karaf.base} property to be correctly
```

```
resolved -->
    <ext:property-placeholder placeholder-prefix="$["
placeholder-suffix="]"/>

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

</blueprint>
```

```
<jaas:keystore xmlns:jaas="http://karaf.apache.org/xmlns/jaas/
v1.1.0"
               name="ks"
               rank="1"
               path="classpath:privatestore.jks"
               keystorePassword="keyStorePassword"
               keyPasswords="myalias=myAliasPassword">
</jaas:keystore>
```

The `id` attribute is the blueprint id of the bean, but it will be used by default as the name of the realm if no `name` attribute is specified. Additional attributes on the `config` elements are a `rank`, which is an integer. When the LoginContext looks for a realm for authenticating a given user, the realms registered in the OSGi registry are matched against the required name. If more than one realm is found, the one with the highest rank will be used, thus allowing the override of some realms with new values. The last attribute is `publish` which can be set to false to not publish the realm in the OSGi registry, hereby disabling the use of this realm.

Each realm can contain one or more module definition. Each module identify 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 will perform 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"`.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/
blueprint-ext/v1.0.0">

    <!-- Bean to allow the $[karaf.base] property to be correctly
resolved -->
    <ext:property-placeholder placeholder-prefix="$["
placeholder-suffix="]"/>

    <jaas:config name="karaf" rank="1">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                     flags="required">
            users = $[karaf.base]/etc/users.properties
            ...
        </jaas:module>
    </jaas:config>

</blueprint>
```

ARCHITECTURE

Due to constraints in the JAAS specification, one class has to be available for all bundles. This class is called ProxyLoginModule and is a LoginModule that acts as a proxy for an OSGi defines LoginModule. If you plan to integrate this feature into another OSGi runtime, this class must be made available from the system classloader and the related package be part of the boot delegation classpath (or be deployed as a fragment attached to the system bundle).

The xml schema defined above allow 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.

AVAILABLE REALMS

Karaf comes with several login modules that can be used to integrate into your environment.

PropertiesLoginModule

This login module is the one configured by default. It uses a properties text file to load the users, passwords and roles from.

Name	Description
`users`	location of the properties file

This file uses the properties file format.
The format of the properties are as follows, each line defining a user, its password and the associated roles:

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

```
<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>
```

OsgiConfigLoginModule

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`.

JDBCLoginModule

The JDBCLoginModule uses a database to load the users, passwords and roles from, provided a data source *(normal or XA)*. The data source and the queries for password and role retrieval are configurable, with the use of 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. See the example 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>
```

LDAPLoginModule

The LDAPLoginModule uses a LDAP to load the users and roles, 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).
`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 server. The default is com.sun.jndi.ldap.LdapCtxFactory

`ssl`	If "true" or if the protocol on the `connection.url` is `ldaps`, an SSL 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 deployed using a `jaas:keystore` configuration.
`ssl.keyalias`	The key alias to use for SSL
`ssl.truststore`	The trust store name to use for SSL. The trust store must be deployed using a `jaas:keystore` configuration.

A example of LDAPLoginModule usage follows:

```
<jaas:config name="karaf">
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
        connection.url = ldap://localhost:389
        user.base.dn = ou=user,dc=apache,dc=org
        user.filter = (cn=%u)
        user.search.subtree = true
        role.base.dn = ou=group,dc=apache,dc=org
        role.filter = (member:=uid=%u)
        role.name.attribute = cn
        role.search.subtree = true
        authentication = simple
  </jaas:module>
</jaas:config>
```

If you want 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" />
```

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.

To

Jasypt

Karaf default installation comes with a simple encryption service which usually fullfill simple needs. However, in some cases, you may want to install the Jasypt library which gives you stronger encryption algorithm and more control over it.

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>
```

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 roles principals by its prefix. 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 = prefix
        role.discriminator = ROLE_
    </jaas:module>
</jaas:config>
```

The application can identify the role principals using a snippet like this:

```
LoginContext ctx = new LoginContext("karaf", handler);
ctx.login();
authenticated = true;
subject = ctx.getSubject();
for (Principal p : subject.getPrincipals()) {
           if (p.getName().startsWith("ROLE_")) {

roles.add((p.getName().substring("ROLE_".length())));
           }
}
```

Grouped Roles

When the group role policy is used the login module provides all roles as members of a group with a configurable name *(property role.discriminator)*. Example:

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

```
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                  flags="required">
        users = $[karaf.base]/etc/users.properties
        role.policy = group
        role.discriminator = ROLES
    </jaas:module>
</jaas:config>
```

```
LoginContext ctx = new LoginContext("karaf", handler);
ctx.login();
authenticated = true;
subject = ctx.getSubject();
for (Principal p : subject.getPrincipals()) {
    if ((p instanceof Group) &&
("ROLES".equalsIgnoreCase(p.getName()))) {
        Group g = (Group) p;
        Enumeration<? extends Principal> members = g.members();
        while (members.hasMoreElements()) {
            Principal member = members.nextElement();
            roles.add(member.getName());
        }
    }
}
```

Writing integration tests

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

Karaf provides an helper library to help writing such integration tests.

```
@Configuration
public static Option[] configuration() throws Exception{
    return combine(
        // Default karaf environment
        Helper.getDefaultOptions(),
        // Test on both equinox and felix
        equinox(), felix()
    );
}
```

If you need to provision a few features in addition to the default karaf environment, you can do so by adding the following code:

```
        scanFeatures(
               maven().groupId("org.apache.felix.karaf")
                      .artifactId("apache-felix-karaf")
                      .type("xml").classifier("features")
                      .versionAsInProject(),
               "obr", "wrapper"
        ),
```