Apache Karaf Container 4.x - Documentation

Apache Software Foundation

Apache Karaf Container 4.x - Documentation

---

1. Overview

Apache Karaf is a modern polymorphic application container.

Karaf can be used as a standalone container, supporting a wide range of applications and technologies. It also supports the "run anywhere" concept (on any machine with Java, cloud, docker images, ...) using the embedded mode.

It's a lightweight, powerful, and enterprise ready platform.

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

Apache Karaf is powered by OSGi (but you don't need to know what OSGi is to use Karaf).

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

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

Here is a short list of provided features:

- **Hot deployment**: simply drop a file in the `deploy` directory, Apache Karaf will detect the type of the file and try to deploy it.

- **Complete Console**: Apache Karaf provides a complete Unix-like console where you can completely manage the container.
- **Dynamic Configuration**: Apache Karaf provides a set of commands focused on managing its own configuration. All configuration files are centralized in the `etc` folder. Any change in a configuration file is noticed and reloaded.
- **Advanced Logging System**: Apache Karaf supports all the popular logging frameworks (slf4j, log4j, etc). Whichever logging framework you use, Apache Karaf centralizes the configuration in one file.
- **Provisioning**: Apache Karaf supports a large set of URLs where you can install your applications (Maven repository, HTTP, file, etc). It also provides the concept of "Karaf Features" which is a way to describe your application.
- **Management**: Apache Karaf is an enterprise-ready container, providing many management indicators and operations via JMX.
- **Remote**: Apache Karaf embeds an SSHd server allowing you to use the console remotely. The management layer is also accessible remotely.
- **Security**: Apache Karaf provides a complete security framework (based on JAAS), and provides a RBAC (Role-Based Access Control) mechanism for console and JMX access.
- **Instances**: multiple instances of Apache Karaf can be managed directly from a main instance (root).
- **OSGi frameworks**: Apache Karaf is not tightly coupled to one OSGi framework. By default, Apache Karaf runs with the Apache Felix Framework, but you can easily switch to Equinox (just change one property in a configuration file).

karaf

2. Quick Start

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

2.1. Prerequisites

Karaf requires a Java SE 8 or higher to run. Refer to http://www.oracle.com/technetwork/java/javase/ for details on how to download and install Java SE 1.8 or greater.

- Open a Web browser and access the following URL: http://karaf.apache.org/download.html

- Download the binary distribution that matches your system (zip for windows, tar.gz for unixes)
- Extract the archive to a new folder on your hard drive; for example in c:\karaf - from now on this directory will be referenced as <KARAF_HOME>.

> **NOTE** Windows users should use 7zip or other unzip tool to support files longer than 255 characters.

2.2. Start the server

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

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

```
bin\karaf.bat
```

respectively on Unix:

```
bin/karaf
```

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

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

  Apache Karaf (4.2.0)

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

karaf@root()>
```

2.3. Shell console basics

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

```
karaf@root()>
karaf: do you wish to see to see all 356 possibilities (219 lines)?
karaf@root()> Display all 294 possibilities? (y or n)
...
shell:logout                        shell:more
shell:new                           shell:printf
shell:sleep                         shell:sort                          shell:source
shell:stack-traces-print            shell:tac
shell:tail                          shell:threads
shell:watch                         shell:wc                            shell:while
shutdown                            sleep
sort                                source
ssh                                 ssh                                 ssh-host-change
ssh-port-change                     ssh:ssh                             stack-traces-
print                  start                              start-
level                         status                             stop
su                                  sudo
system                              system:framework
system:name                         system:property                     system:shutdown
system:start-level                  system:version
tac                                 tail
threads                             tree-show                           uninstall
update                              user-add                            user-
delete                         user-list
version                             version-list                        wait
watch                               wc                                  while
```

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

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

        Lists all installed bundles.

SYNTAX
        bundle:list [options] [ids]

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

OPTIONS
        -name, -n
                Show bundle name
        -u
                Shows the update locations
        -r
                Shows the bundle revisions
        --no-ellipsis

        -l
                Show the locations
        -s
                Shows the symbolic name
        --context, -c
                Use the given bundle context
                (defaults to 0)
        --help
                Display this help message
        -t
                Specifies the bundle threshold; bundles with a start-level less than this
value will not get printed out.
        --no-format
                Disable table rendered output
```

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

2.4. Deploy a sample application

While you will learn in the Karaf user's guide how to fully use and leverage Apache Karaf, let's install a sample Apache Camel (http://camel.apache.org) application for now:

Copy and paste the following commands in the console:

```
feature:repo-add camel
feature:install deployer camel-blueprint aries-blueprint
cat > deploy/example.xml <<END
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <from uri="timer://test?fixedRate=true&amp;period=2000" />
            <setBody>
                <simple>Message at ${date:now:yyyy-MM-dd HH:mm:ss}</simple>
            </setBody>
            <to uri="log:test" />
        </route>
    </camelContext>

</blueprint>
END
```

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

You can display the log in the shell:

```
karaf@root()> log:display
...
2017-10-13 10:04:11,225 | INFO  | 7 - timer://test | test                                 |
28 - org.apache.camel.camel-core - 2.20.0 | Exchange[ExchangePattern: InOnly, BodyType:
String, Body: Message at 2017-10-13 10:04:11]
2017-10-13 10:04:13,225 | INFO  | 7 - timer://test | test                                 |
28 - org.apache.camel.camel-core - 2.20.0 | Exchange[ExchangePattern: InOnly, BodyType:
String, Body: Message at 2017-10-13 10:04:13]
```

2.5. Stopping and uninstalling the sample application

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

```
karaf@root()> bundle:stop example.xml
karaf@root()> bundle:uninstall example.xml
```

2.6. Stopping Karaf

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

```
^D
```

Alternatively, you can also run the following command:

```
karaf@root()> feature:install system
karaf@root()> system:shutdown
Confirm: halt instance root (yes/no): yes
karaf@root()>
```

halt is also an alias for system:shutdown:

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

2.7. Cleaning the Karaf state

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

2.8. Summary

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

3. Update Notes (from Karaf 3.x to 4.x versions)

This section is dedicated to users of previous Apache Karaf version.

NOTE	For the users upgrading from Karaf 2.x, please see the "Update Notes (from 2.x to 3.x)" in the Karaf 3.x documentation first.

NOTE	Karaf 4.x supports Java8.

3.1. Distributions

Apache Karaf 4.x is available as tar.gz and zip archives. The content is the same and works on either Unix or Windows platforms. The `bin` folder contains both sh and bat scripts.

3.2. Commands

Karaf 4 commands are close to the ones provided in Karaf 3.

The sub-shell modes are the same (in `etc/org.apache.karaf.shell.cfg` or using `shell:completion` command).

However, some minor changes have been introduced and new commands available:

Apache Karaf 4.x
feature:requirement-list
feature:requirement-add
feature:requirement-remove
feature:regions
feature:start
feature:stop
jaas:group-create
jaas:group-add
jaas:group-delete
jaas:group-list
jaas:group-role-add
jaas:group-role-delete
jaas:su
jaas:sudo
shell:edit
shell:env

Apache Karaf 4.x
shell:less
shell:stack-traces-print
shell:threads
shell:while
log:list
bundle:capabilities
bundle:diag
bundle:id
bundle:load-test
bundle:requirements
bundle:resolve
system:name

We encourage the users to use the `--help` option to check the name and type of arguments and options.

In term of development, you can still use the blueprint definition as you do in Karaf 2.x & 3.x (with the corresponding annotations).

However, in Karaf 4.x, you can use DS and new annotations and avoid the usage of a blueprint XML.

The new annotations are available: @Service, @Completion, @Parsing, @Reference. It allows you to completely define the command in the command class directly.

To simplify the generation of the code and OSGi headers, Karaf 4.x provides the karaf-services-maven-plugin (in org.apache.karaf.tooling Maven groupId).

Take a look in the developer guide for the command development "new style" details.

3.3. Features repositories

Karaf 4.x provides:

- `mvn:org.apache.karaf.features/enterprise/4.0.x/xml/features`
- `mvn:org.apache.karaf.features/framework/4.0.x/xml/features`
- `mvn:org.apache.karaf.features/spring/4.0.x/xml/features`
- `mvn:org.apache.karaf.features/standard/4.0.x/xml/features`
- `mvn:org.apache.karaf.features/static/4.0.x/xml/features`

3.4. Features resolver

Apache Karaf 4.x brings a complete new feature resolver. The purpose is to simplify the features installation and lifecycle. The new resolver now checks the feature requirements (defined directly in the features XML), and check which bundles provides the capabilities to satisfy these requirements. It allows Karaf to automatically install bundles required by features.

For "old style" feature (features XML using namespace from Karaf 2.x or 3.x), this feature is not enabled. As soon as you use a "new style" feature (with namespace 1.3.0 used by Karaf 4.x), this feature is enabled.

3.5. Namespaces

Apache Karaf 4.x brings updated version of the namespaces:

- Supported features namespaces:
  - `karaf-features-1.0.0.xsd`
  - `karaf-features-1.1.0.xsd`
  - `karaf-features-1.2.0.xsd`
  - `karaf-features-1.2.1.xsd`
  - `karaf-features-1.3.0.xsd`
- Supported jaas namespaces:
  - `karaf-jaas-1.0.0.xsd`

    - `karaf-jaas-1.1.0.xsd`
- Supported shell namespaces:
    - `karaf-shell-1.0.0.xsd`
    - `karaf-shell-1.1.0.xsd`

3.6. Maven plugin

A cleanup of the goals provided by {{karaf-maven-plugin}} has been done.

Now the provided goals are:

- `karaf:archive` to create a tar.gz or zip of a Karaf distribution
- `karaf:assembly` to create a custom Karaf distribution assembly
- `karaf:kar` to create a kar file
- `karaf:verify` to verify and validate Karaf features
- `karaf:features-add-to-repository` to recursively copy features XML and content into a folder (repository)
- `karaf:features-export-meta-data` to extract the metadata from a features XML
- `karaf:features-generate-descriptor` to generate a features XML
- `karaf:commands-generate-help` to generate help/documentation on the commands
- `karaf:run` to run a Karaf container directly from Maven
- `karaf:client` to interact with a remote Karaf instance
- `karaf:deploy` to deploy an application to a remote Karaf instance

3.7. Update guide

We encourage users to start a fresh Apache Karaf 4.x container.

If you upgrade an existing container, `lib` and `system` folder have to be updated (just an override copy).

For the `etc` folder, a diff is required as some properties have changed and new configurations are available.

4. User Guide

4.1. Installation

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

4.1.1. Requirements

Hardware:

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

Operating Systems:

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

Environment:

- Java SE 1.8 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,

4.1.2. Using Apache Karaf binary distributions

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

The "default" distribution is a "ready to use" distribution, with pre-installed features.

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

Installation on Windows platform

NOTE

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

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

NOTE

Remember the restrictions concerning illegal characters in Java paths, e.g. \!, % etc.

NOTE

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

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

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

Installation on Unix platforms

NOTE

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

```
echo $JAVA_HOME
```

If it's not correct, fix it using:

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

1. From a browser, navigate to http://karaf.apache.org/download.html.

2. Download Apache Karaf binary distribution in the tar.gz format: `apache-karaf-4.0.0.tar.gz`.
3. Extract the files from the tar.gz file into a directory of your choice (it's the `KARAF_HOME`). For example:

```
tar zxvf apache-karaf-4.0.0.tar.gz
```

> **NOTE** Remember the restrictions concerning illegal characters in Java paths, e.g. \!, % etc.

Installation on Debian GNU/linux

There is an unofficial debian package of karaf:

1. Add the key of the unofficial debian archive containing karaf

```
wget -O - https://apt.bang.priv.no/apt_pub.gpg | apt-key add -
```

2. Open /etc/apt/sources.list in a text editor and add the following lines:

```
# Unofficial APT archive for apache karaf
deb http://apt.bang.priv.no/public stable main
```

3. Update the APT package index and install the package

```
apt-get update
apt-get install openjdk-8-jdk karaf
```

The installed karaf will run as user karaf, group karaf, listening to the usual default karaf ports, be started with systemd, with KARAF_ETC in /etc/karaf and KARAF_DATA in /var/lib/karaf/data.

> **NOTE** When/if the link: `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=881297` [Debian Request For Package (RFP) of karaf] results in an official karaf package becoming available, the unofficial debian package will be discontinued.

4.1.3. Post-Installation steps

Thought it is not always required, it is strongly advised to set up the `JAVA_HOME` environment property to point to the JDK you want Apache Karaf to use before starting it. This property is used to locate the `java` executable and should be configured to point to the home directory of the Java SE 8 installation.

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

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

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

4.1.4. Building from Sources

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

Hardware:

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

Environment:

- Java SE Development Kit 1.8 or greater (http://www.oracle.com/technetwork/java/javase/).
- Apache Maven 3.5.0 or greater (http://maven.apache.org/download.html).

Building on Windows platform

You can get the Apache Karaf sources from:

- the sources distribution `apache-karaf-4.0.0-src.zip` available at http://karaf.apache.org/download.html. Extract the files in the directory of your choice.
- by checking out the git repository:

```
git clone https://github.com/apache/karaf karaf
```

Use Apache Maven to build Apache Karaf:

```
mvn clean install
```

NOTE

You can speed up the build by bypassing the unit tests:

```
mvn clean install -DskipTests
```

Now, you can find the built binary distribution in `assemblies\apache-karaf\target\apache-karaf-4.0.0.zip`.

Building on Unix platforms

You can get the Apache Karaf sources from:

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

```
git clone https://github.com/apache/karaf karaf
```

Use Apache Maven to build Apache Karaf:

```
mvn clean install
```

NOTE

You can speed up the build by bypassing the unit tests:

```
mvn clean install -DskipTests
```

Now, you can find the built binary distribution in `assemblies/apache-karaf/target/apache-karaf-4.0.0.tar.gz`.

4.2. Directory structure

The directory layout of a Karaf installation is as follows:

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

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

4.3. Start, stop, restart, connect

4.3.1. Start

Apache Karaf supports different start modes:

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

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

Regular mode

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

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

On Unix:

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

  Apache Karaf (4.0.0)

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

karaf@root()>
```

On Windows:

```
bin\karaf.bat
        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

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

karaf@root()>
```

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

Server mode

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

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

On Unix:

```
bin/karaf server
```

On Windows:

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

> **NOTE** Closing the console or shell window will cause Apache Karaf to terminate.

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

Background mode

The background mode starts Apache Karaf as a background process.

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

On Unix:

```
bin/start
```

On Windows:

```
bin\start.bat
```

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

Clean start

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

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

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

On Unix:

```
bin/karaf clean
```

```
bin/start clean
```

On Windows:

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

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

Customize variables

Apache Karaf accepts environment variables:

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

- `KARAF_OPTS`: extra arguments passed to the Java command line (default is null).
- `KARAF_DEBUG`: if *true*, enable the debug mode (default is null). If debug mode is enabled, Karaf starts a JDWP socket on port 5005. You can plug your IDE to define breakpoints, and run step by step.

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

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

On Unix:

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

On Windows:

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

Connect

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

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

On Unix:

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

  Apache Karaf (4.0.0)

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

karaf@root()>
```

On Windows:

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

  Apache Karaf (4.0.0-SNAPSHOT)

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

karaf@root()>
```

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

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

On Unix:

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

On Windows:

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

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

For instance, on Unix, you can do:

```
ssh karaf@localhost -p 8101
Authenticated with partial success.
Authenticated with partial success.
Authenticated with partial success.
Password authentication
Password:
        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0-SNAPSHOT)

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

karaf@root()>
```

4.3.2. Stop

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

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

More generally, you can use the `shutdown` command (on the Apache Karaf console) that works for all cases.

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

To shutdown Apache Karaf now, you can simply use `shutdown`:

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

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

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

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

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

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

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

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

Another example to shutdown Apache Karaf in 10 minutes:

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

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

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

        Shutdown Karaf.

SYNTAX
        system:shutdown [options] [time]

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

OPTIONS
        -c, --clean, --clean-all, -ca
                Force a clean restart by deleting the data directory
        -f, --force
                Force the shutdown without confirmation message.
        -h, --halt
                Halt the Karaf container.
        --help
                Display this help message
        -cc, --clean-cache, -cc
                Force a clean restart by deleting the cache directory
        -r, --reboot
                Reboot the Karaf container.
```

4.3.3. Status

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

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

> **NOTE** The status script returns 0 exit code if Apache Karaf is running, 1 exit code else.

On Unix:

```
bin/status
Not Running ...
```

```
bin/status
Running ...
```

On Windows:

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

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

4.3.4. Restart

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

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

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

4.3.5. SystemMBean

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

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

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

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

- `halt(time)` shutdown Apache Karaf at a given time. The time format is the same as the time argument of the `shutdown` command.

4.4. Integration in the operating system

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

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

- Service Wrapper
- Service Script Templates

The above methods allow you to directly integrate Apache Karaf:

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

4.4.1. Service Wrapper

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

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

Supported platforms

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

- MacOS X

Installation

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

In the console:

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

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

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

        Install the container as a system service in the OS.

SYNTAX
        wrapper:install [options]

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

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

For instance, on a Ubuntu/Debian Linux system:

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

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


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

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

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

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

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

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

NOTE

You can install the wrapper without starting Karaf using bin/shell wrapper:install command.

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

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

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

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

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

This will enable Karaf at system boot.

Uninstall

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

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

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

If you preferred the systemd service instead of systemV:

```
systemctl disable karaf
```

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

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

Note for MacOS users

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

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

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

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

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

Change owner and rights:

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

You can test your service:

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

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

If you want to remove the service call:

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

Configuration

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

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

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

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

- `set.default.JAVA_HOME` is the `JAVA_HOME` used to start Apache Karaf (populated during Service Wrapper installation).
- `set.default.KARAF_HOME` is the location of your Apache Karaf installation (populated during Service Wrapper installation).
- `set.default.KARAF_BASE` is the location of your Apache Karaf installation (populated during Service Wrapper installation).
- `set.default.KARAF_DATA` is the location of the Apache Karaf data folder (populated during Service Wrapper installation).
- `set.default.KARAF_ETC` is the location of the Apache Karaf etc folder (populated during Service Wrapper installation).
- `wrapper.java.additional` is used to pass additional arguments to the Java command, indexed by the argument number. The next index to use is 11.
- `wrapper.java.initmemory` is the initial JVM memory size (the `-Xms`). It's not set by default (JVM default).
- `wrapper.java.maxmemory` is the maximum JVM memory size (the `-Xmx`). It's set to 512M by default.
- `wrapper.logfile` is the location of the Service Wrapper log file. It's set to `%KARAF_DATA%/log/wrapper.log` by default.
- `wrapper.logfile.loglevel` is the Service Wrapper log level. It's set to `INFO` by default.
- `wrapper.logfile.maxsize` is the Service Wrapper log file maximum size (before rotation). It's set to `10m` (10MB) by default.
- `wrapper.logfile.maxfiles` is the number of Service Wrapper log files created (and rotated). It's set to `5` by default.
- `wrapper.syslog.loglevel` is the integration with Unix syslog daemon. By default, it's set to `none` meaning disabled.
- `wrapper.ntservice.name` is Windows service specific and defines the Windows service name. It's set to the `name` option of the `wrapper:install` command, or `karaf` by default.

- `wrapper.ntservice.displayname` is Windows service specific and defines the Windows service display name. It's set to the `display` option of the `wrapper:install` command, or `karaf` by default.
- `wrapper.ntservice.description` is Windows service specific and defines the Windows service description. It's set to the `description` option of the `wrapper:install` command, or empty by default.
- `wrapper.ntservice.starttype` is Windows service specific and defines if the Windows service is started automatically with the service, or just on demand. It's set to `AUTO_START` by default, and could be switched to `DEMAND_START`.

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

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

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

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

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

# JVM Parameters
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dkaraf.home=%KARAF_HOME%
wrapper.java.additional.2=-Dkaraf.base=%KARAF_BASE%
wrapper.java.additional.3=-Dkaraf.data=%KARAF_DATA%
wrapper.java.additional.4=-Dkaraf.etc=%KARAF_ETC%
wrapper.java.additional.5=-Dkaraf.startLocalConsole=false
wrapper.java.additional.6=-Dkaraf.startRemoteShell=true
wrapper.java.additional.7=-Djava.io.tmpdir=%KARAF_DATA%/tmp
```

```
# Uncomment if you need libraries on Java 8
#wrapper.java.additional.9=-Djava.endorsed.dirs=%JAVA_HOME%/jre/lib/
endorsed:%JAVA_HOME%/lib/endorsed:%KARAF_HOME%/lib/endorsed
#wrapper.java.additional.10=-Djava.ext.dirs=%JAVA_HOME%/jre/lib/ext:%JAVA_HOME%/lib/
ext:%KARAF_HOME%/lib/ext

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

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

# Uncomment to enable remote debugging
#wrapper.java.additional.n=-Xdebug -Xnoagent -Djava.compiler=NONE
#wrapper.java.additional.n=-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005

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

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


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

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

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

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

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

# Maximum size that the log file will be allowed to grow to before
#  the log is rolled. Size is specified in bytes.  The default value
#  of 0, disables log rolling.  May abbreviate with the 'k' (kb) or
#  'm' (mb) suffix.  For example: 10m = 10 megabytes.
wrapper.logfile.maxsize=10m
```

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

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

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

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

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

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

# Description of the service
wrapper.ntservice.description=

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

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

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

4.4.2. Service Script Templates

By using the "Service Script Templates", you can run Apache Karaf with the help of operating system specific init scripts.

> **NOTE** As opposed to the Service Wrapper, the templates targeting Unix systems do not rely on 3rd party binaries

You can find these templates under the bin/contrib directory.

Unix

The karaf-service.sh utility helps you to generate ready to use scripts by automatically identifying the operating system, the default init system and the template to use.

> **NOTE** You may still need to customize the generated files to adapt them to your environment.

The utility karaf-service.sh can be configured by defining environment variables or by passing command line options:

Command line option	Environment variable	Description
-k	KARAF_SERVICE_PATH	Karaf installation path (mandatory)
-d	KARAF_SERVICE_DATA	Karaf data path (default to ${KARAF_SERVICE_PATH}/data)
-c	KARAF_SERVICE_CONF	Karaf configuration file (default to ${KARAF_SERVICE_PATH/ etc/${KARAF_SERVICE_NAME}.conf
-t	KARAF_SERVICE_ETC	Karaf etc path (default to ${KARAF_SERVICE_PATH/etc}
-p	KARAF_SERVICE_PIDFILE	Karaf pid path (default to ${KARAF_SERVICE_DATA}/${KARAF_SERVICE_NAME}.p
-n	KARAF_SERVICE_NAME	Karaf service name (default karaf)
-e	KARAF_ENV	Karaf environment variable (can be repeated)
-u	KARAF_SERVICE_USER	Karaf user
-g	KARAF_SERVICE_GROUP	Karaf group (default ${KARAF_SERVICE_USER)

-l	KARAF_SERVICE_LOG	Karaf console log (default to ${KARAF_SERVICE_DATA}/log/${KARAF_SERVICE_NAM console.log)
-f	KARAF_SERVICE_TEMPLATE	Template file to use
-x	KARAF_SERVICE_EXECUTABLE	Karaf executable name (defaul karaf

Systemd

When karaf-service.sh detects Systemd, it generates three files:

- a systemd unit file to manage the root Apache Karaf container
- a systemd environment file with variables used by the root Apache Karaf container
- a systemd template unit file to manage Apache Karaf child containers

Example

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.service"
Writing service configuration file ""/opt/karaf-4/etc/karaf-4.conf"
Writing service file "/opt/karaf-4/bin/contrib/karaf-4@.service"

$ cp /opt/karaf-4/bin/contrib/karaf-4.service /etc/systemd/system
$ cp /opt/karaf-4/bin/contrib/karaf-4@.service /etc/systemd/system

$ systemctl enable karaf-4.service
```

SysV

When karaf-service.sh detects a SysV system, it generates two files:

- an init script to manage the root Apache Karaf container
- an environment file with variables used by the root Apache Karaf container

Example

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4"
Writing service configuration file "/opt/karaf-4/etc/karaf-4.conf"

$ ln -s /opt/karaf-4/bin/contrib/karaf-4 /etc/init.d/
$ chkconfig karaf-4 on
```

NOTE | To enable service startup upon boot, please consult your operating system init guide

Solaris SMF

When karaf-service.sh detects a Solaris system, it generates a single file:

Example

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4 -u lburgazz -g lburgazz
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.xml"

$ svccfg validate /opt/karaf-4/bin/contrib/karaf-4.xml
$ svccfg import /opt/karaf-4/bin/contrib/karaf-4.xml
```

NOTE | The generated SMF descriptor is defined as transient so the start method is executed once

Windows

Installation of Apache Karaf as windows service is supported through winsw (https://github.com/kohsuke/winsw).

Steps:

- Rename karaf-service-win.exe to the service name i.e karaf-4.exe
- Rename karaf-service-win.xml to match the service name i.e. karaf-4.xml
- Customize the service descriptor to fit your needs
- Use the service executable to install/star/stop the service

Example

```
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe install
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe start
```

4.5. Using the console

4.5.1. Available commands

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

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

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

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

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

4.5.2. Subshell and completion mode

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

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

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

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

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

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

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

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

The prompt displays the current subshell between ().

The `exit` command goes to the parent subshell:

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

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

You have three different modes available:

- GLOBAL
- FIRST
- SUBSHELL

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

```
completionMode = GLOBAL
```

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

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

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

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

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

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

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

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

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

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

SUBSHELL completion mode is the real subshell mode.

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

```
karaf@root()> shell:completion SUBSHELL
karaf@root()> <TAB>
karaf@root()>
* bundle cl config dev feature help instance jaas kar la ld lde log log:list man package
region service shell ssh system
karaf@root()> bundle
karaf@root(bundle)> <TAB>
karaf@root(bundle)>
capabilities classes diag dynamic-import find-class headers info install list refresh
requirements resolve restart services start start-level stop
uninstall update watch
karaf@root(bundle)> exit
karaf@root()> camel
karaf@root(camel)> <TAB>
karaf@root(camel)>
backlog-tracer-dump backlog-tracer-info backlog-tracer-start backlog-tracer-stop context-
info context-list context-start context-stop endpoint-list route-info route-list route-
profile route-reset-stats
route-resume route-show route-start route-stop route-suspend
```

4.5.3. Unix like environment

Karaf console provides a full Unix like environment.

Help or man

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

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

So these commands

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

All produce the same help output:

```
DESCRIPTION
        feature:list

        Lists all existing features available from the defined repositories.

SYNTAX
        feature:list [options]

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

Completion

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

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

Alias

An alias is another name associated to a given command.

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

```
karaf@root()> alias "list-features-installed = { feature:list -i }"
karaf@root()> list-features-installed
Name       | Version | Required | State   | Repository     | Description
------------------------------------------------------------------------------------------
-----------------------------------
feature    | 4.0.0   | x        | Started | standard-4.0.0 | Features Support
shell      | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Shell
deployer   | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Deployer
bundle     | 4.0.0   | x        | Started | standard-4.0.0 | Provide Bundle support
config     | 4.0.0   | x        | Started | standard-4.0.0 | Provide OSGi ConfigAdmin
support
diagnostic | 4.0.0   | x        | Started | standard-4.0.0 | Provide Diagnostic support
instance   | 4.0.0   | x        | Started | standard-4.0.0 | Provide Instance support
jaas       | 4.0.0   | x        | Started | standard-4.0.0 | Provide JAAS support
log        | 4.0.0   | x        | Started | standard-4.0.0 | Provide Log support
package    | 4.0.0   | x        | Started | standard-4.0.0 | Package commands and mbeans
service    | 4.0.0   | x        | Started | standard-4.0.0 | Provide Service support
system     | 4.0.0   | x        | Started | standard-4.0.0 | Provide System support
kar        | 4.0.0   | x        | Started | standard-4.0.0 | Provide KAR (KARaf archive)
support
ssh        | 4.0.0   | x        | Started | standard-4.0.0 | Provide a SSHd server on
Karaf
management | 4.0.0   | x        | Started | standard-4.0.0 | Provide a JMX MBeanServer
and a set of MBeans in
```

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

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

You can see here the aliases available by default:

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

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

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

Key binding

Like on most Unix environments, the Karaf console supports some key bindings:

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

Pipe

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

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

Grep, more, find, ...

Karaf console provides some core commands similar to a Unix environment:

- `shell:alias` creates an alias to an existing command
- `shell:cat` displays the content of a file or URL

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

- `shell:watch` periodically executes a command and refresh the output
- `shell:wc` prints newline, words, and byte counts for each file
- `shell:while` loop while the condition is true

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

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

Scripting

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

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

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

NOTE

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

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

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

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

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

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

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

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

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

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


Signers              null
SimpleName           ArrayList
TypeParameters       [E]
```

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

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

If it fails, you will see the casting exception:

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

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

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

where `script.txt` contains:

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

NOTE

The spaces are important when writing scripts. For instance, the following script is not correct:

```
if{ $foo equals "foo" } ...
```

and will fail with:

```
karaf@root> shell:source script.txt
Error executing command: Cannot coerce echo "true!"() to any of []
```

because a space is missing after the `if` statement.

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

See the Scripting section of the developers guide for details.

4.5.4. Security

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

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

4.6. Remote

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

Apache Karaf embeds a complete SSHd server.

NOTE

For security reason, by default, `karaf` user is disabled. To allow the logon, you have to have an user. You can enable `karaf` user by uncommenting the user in `etc/users.properties`:

```
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

4.6.1. SSHd server

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

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

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

Configuration

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

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

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

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

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

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

#
# The location of the hostKey file defines where the private key of the server
# is located. If no file is at the defined location it will be ignored.
#
hostKey = ${karaf.etc}/host.key

#
# The password required to decrypt the private key of the server stored in
# 'hostKey'. This is not required if the private key stored in 'hostKey' is
```

```
# not encrypted
#hostKeyPassword =

#
# The location of the hostKeyPub file defines where the public key of the server
# is located. If no file is at the defined location it will be ignored.
#
#hostKeyPub = ${karaf.etc}/host.key.pub

#
# sshRole defines the role required to access the console through ssh
#
# sshRole = ssh

#
# Defines if the SFTP system is enabled or not in the SSH server
#
sftpEnabled=true

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

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

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

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

- `sshPort` is the port number where the SSHd server is bound (by default, it's 8101).
- `sshHost` is the address of the network interface where the SSHd server is bound. The default value is 0.0.0.0, meaning that the SSHd server is bound on all network interfaces. You can bind on a target interface by providing the IP address of the network interface.
- `hostKey` is the location of the `host.key` file. By defaut, it uses `etc/host.key`. This file stores the private key of the SSHd server.
- `hostKeyPassword` is the password required to decrypt the private key of the server stored in `hostKey`, if the key is stored in an encrypted form. Note that Karaf does not use this property to encrypt the private key when generating it, only for reading external keys that are already encrypted. Also note that specifying a `hostKeyPassword` might require installing the BouncyCastle provider to support the desired encryption algorithm.
- `hostKeyPub` is the location of the public key of the server.
- `sshRole` is the default role used for SSH access. See the [Security section|security] of this user guide for details.
- `sftpEnabled` controls if the SSH server starts the SFTP system or not. When enabled, Karaf SSHd supports SFTP, meaning that you can remotely access the Karaf filesystem with any sftp client.
- `keySize` is the key size used by the SSHd server. The possible values are 1024, 2048, 3072, or 4096. The default value is 2048.
- `algorithm` is the host key algorithm used by the SSHd server. The possible values are DSA, EC or RSA. The default value is RSA.

The SSHd server configuration can be changed at runtime:

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

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

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

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

Console clients

System native clients

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

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

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

```
~$ ssh -p 8101 karaf@localhost
Authenticated with partial success.
Authenticated with partial success.
Authenticated with partial success.
Password authentication
Password:
        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

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

karaf@root()>
```

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

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

`ssh:ssh` command

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

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

        Connects to a remote SSH server

SYNTAX
        ssh:ssh [options] hostname [command]

ARGUMENTS
        hostname
                The host name to connect to via SSH
        command
                Optional command to execute

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

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

```
karaf@root()> ssh:ssh -p 8101 karaf@192.168.134.2
Connecting to host 192.168.134.2 on port 8101
Connecting to unknown server. Add this server to known hosts ? (y/n)
Storing the server key in known_hosts.
Connected
        __ __                  ____
       / //_/____ _____________ _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

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

karaf@root()>
```

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

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

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

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

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

Apache Karaf client

The `ssh:ssh` command can only be run in a running Apache Karaf console.

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

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

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

```
bin/client
Logging in as karaf
343 [pool-2-thread-4] WARN org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier
- Server at /0.0.0.0:8101 presented unverified key:
        __ __                  ____
       / //_/____ _____ _____ / __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.0.0)

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

karaf@root()>
```

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

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

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

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

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

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

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

Logout

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

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

Filesystem clients

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

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

Native SCP/SFTP clients

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

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

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

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

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

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

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

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

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

Apache Maven

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

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

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

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

NOTE	If you want to turn Apache Karaf as a simple Maven repository, you can use Apache Karaf Cave (https://karaf.apache.org/projects.html).

4.6.2. JMX MBeanServer

Apache Karaf provides a JMX MBeanServer.

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

You can find details on the [Monitoring section|monitoring] of the user guide.

4.7. Log

Apache Karaf provides a very dynamic and powerful logging system.

It supports:

- the OSGi Log Service
- the Apache Log4j v1 and v2 framework
- the Apache Commons Logging framework
- the Logback framework
- the SLF4J framework
- the native Java Util Logging framework

It means that applications can use any logging framework, Apache Karaf will use the central log system to manage the loggers, appenders, etc.

4.7.1. Configuration files

The initial log configuration is loaded from `etc/org.ops4j.pax.logging.cfg`.

This file is a standard Log4j configuration file (http://logging.apache.org/log4j/1.2/manual.html).

You find the different Log4j elements:

- loggers
- appenders
- layouts

You can add your own initial configuration directly in the file.

The default configuration is the following:

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

# Root logger
log4j.rootLogger=INFO, out, osgi:*
log4j.throwableRenderer=org.apache.log4j.OsgiThrowableRenderer

# CONSOLE appender not used by default
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} | %-5.5p | %-16.16t |
%-32.32c{1} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n

# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1}
| %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.log}/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10

# Sift appender
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=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{ISO8601} | %-5.5p | %-16.16t |
%-32.32c{1} | %m%n
log4j.appender.sift.appender.file=${karaf.log}/$\\{bundle.name\\}.log
log4j.appender.sift.appender.append=true
```

The default configuration only define the `ROOT` logger, with `INFO` log level, using the `out` file appender. You can change the log level to any Log4j valid values (from the most to less verbose): TRACE, DEBUG, INFO, WARN, ERROR, FATAL.

The `osgi:*` appender is a special appender to send the log message to the OSGi Log Service.

A `stdout` console appender is pre-configured, but not enabled by default. This appender allows you to display log messages directly to standard output. It's interesting if you plan to run Apache Karaf in server mode (without console).

To enable it, you have to add the `stdout` appender to the `rootLogger`:

```
log4j.rootLogger=INFO, out, stdout, osgi:*
```

The `out` appender is the default one. It's a rolling file appender that maintains and rotates 10 log files of 1MB each. The log files are located in `data/log/karaf.log` by default.

The `sift` appender is not enabled by default. This appender allows you to have one log file per deployed bundle. By default, the log file name format uses the bundle symbolic name (in the `data/log` folder).

You can edit this file at runtime: any change will be reloaded and be effective immediately (no need to restart Apache Karaf).

Another configuration file is used by Apache Karaf: `etc/org.apache.karaf.log.cfg`. This files configures the Log Service used by the log commands (see later).

4.7.2. Log4j v2 support

Karaf supports log4j v2 backend.

To enable log4j v2 support you have to:

1. Edit `etc/startup.properties` to replace the line `org/ops4j/pax/logging/pax-logging-service/1.8.4/pax-logging-service-1.8.4.jar=8` with `org/ops4j/pax/logging/pax-logging-log4j2/1.8.4/pax-logging-log4j2-1.8.4.jar=8`
2. Add pax-logging-log4j2 jar file in `system/org/ops4j/pax/logging/pax-logging-log4j2/x.x/pax-logging-log4j2-x.x.jar where x.x is the version as defined in `etc/startup.properties`

3. Edit `etc/org.ops4j.pax.logging.cfg` configuration file and add `org.ops4j.pax.logging.log4j2.config.file=${karaf.etc}/log4j2.xml`
4. Add the `etc/log4j2.xml` configuration file.

A default configuration in `etc/log4j2.xml` could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} |
%X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n"/>
        </Console>
        <RollingFile name="out" fileName="${karaf.log}/karaf.log"
              append="true" filePattern="${karaf.log}/$${date:yyyy-MM}/fuse-%d{MM-dd-
yyyy}-%i.log.gz">
           <PatternLayout>
             <Pattern>%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} | %X{bundle.id} -
%X{bundle.name} - %X{bundle.version} | %m%n</Pattern>
           </PatternLayout>
           <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="250 MB"/>
            </Policies>
        </RollingFile>
        <PaxOsgi name="paxosgi" filter="VmLogAppender"/>
    </Appenders>
    <Loggers>
        <Root level="INFO">
            <AppenderRef ref="console"/>
            <AppenderRef ref="out"/>
            <AppenderRef ref="paxosgi"/>
        </Root>
    </Loggers>
</Configuration>
```

4.7.3. `karaf.log.console` property

Before Karaf starts *proper* logging facilities (pax-logging), it may configure `java.util.logging`. Standard Java logging is used initially by the `Main` class and `org.apache.karaf.main.lock.Lock` implementations. In order to configure the logging level, please set the system property `karaf.log.console` to one of the standard JUL levels:

- `SEVERE` (highest value)

- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST` (lowest value)

Additionally, de-facto standard log4j(2) levels can be used:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `OFF`
- `DEFAULT`

And because `org.ops4j.pax.logging` PID uses `karaf.log.console` property, it's in fact better to use log4j levels instead:

```
log4j2.rootLogger.appenderRef.Console.filter.threshold.level = ${karaf.log.console:-OFF}
```

For example, setting `karaf.log.console` to `INFO` (or lower) will turn on these logs when starting Karaf:

```
Jul 04, 2017 7:53:18 AM org.apache.karaf.main.Main launch
INFO: Installing and starting initial bundles
Jul 04, 2017 7:53:18 AM org.apache.karaf.main.Main launch
INFO: All initial bundles installed and set to start
...
```

4.7.4. Commands

Instead of changing the `etc/org.ops4j.pax.logging.cfg` file, Apache Karaf provides a set of commands allowing to dynamically change the log configuration and see the log content:

`log:clear`

The `log:clear` command clears the log entries.

`log:display`

The `log:display` command displays the log entries.

By default, it displays the log entries of the `rootLogger`:

```
karaf@root()> log:display
2015-07-01 19:12:46,208 | INFO  | FelixStartLevel  | SecurityUtils                    |
16 - org.apache.sshd.core - 0.12.0 | BouncyCastle not registered, using the default JCE
provider
2015-07-01 19:12:47,368 | INFO  | FelixStartLevel  | core                             |
68 - org.apache.aries.jmx.core - 1.1.1 | Starting JMX OSGi agent
```

You can also display the log entries from a specific logger, using the `logger` argument:

```
karaf@root()> log:display ssh
2015-07-01 19:12:46,208 | INFO  | FelixStartLevel  | SecurityUtils                    |
16 - org.apache.sshd.core - 0.12.0 | BouncyCastle not registered, using the default JCE
provider
```

By default, all log entries will be displayed. It could be very long if your Apache Karaf container is running since a long time. You can limit the number of entries to display using the `-n` option:

```
karaf@root()> log:display -n 5
2015-07-01 06:53:24,143 | INFO  | JMX OSGi Agent   | core                                 |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.BundleStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=bundleState,version=1.7,framework=org.apache.felix.framework,uuid=5335370f-
9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,150 | INFO  | JMX OSGi Agent   | core                                 |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.PackageStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=packageState,version=1.5,framework=org.apache.felix.framework,uuid=5335370f
-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,150 | INFO  | JMX OSGi Agent   | core                                 |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.ServiceStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=serviceState,version=1.7,framework=org.apache.felix.framework,uuid=5335370f
-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,152 | INFO  | JMX OSGi Agent   | core                                 |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.wiring.BundleWiringStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=wiringState,version=1.1,framework=org.apache.felix.framework,uuid=5335370f-
9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,501 | INFO  | FelixStartLevel  | RegionsPersistenceImpl               |
78 - org.apache.karaf.region.persist - 4.0.0 | Loading region digraph persistence
```

You can also limit the number of entries stored and retained using the `size` property in the `etc/org.apache.karaf.log.cfg` file:

```
#
# The number of log statements to be displayed using log:display. It also defines the
number
# of lines searched for exceptions using log:display exception. You can override this
value
# at runtime using -n in log:display.
#
size = 500
```

By default, each log level is displayed with a different color: ERROR/FATAL are in red, DEBUG in purple, INFO in cyan, etc. You can disable the coloring using the `--no-color` option.

The log entries format pattern doesn't use the conversion pattern defined in `etc/org.ops4j.pax.logging.cfg` file. By default, it uses the `pattern` property defined in `etc/org.apache.karaf.log.cfg`.

```
#
# The pattern used to format the log statement when using log:display. This pattern is
according
# to the log4j layout. You can override this parameter at runtime using log:display with
-p.
#
pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} | %X{bundle.id} - %X{bundle.name}
- %X{bundle.version} | %m%n
```

You can also change the pattern dynamically (for one execution) using the `-p` option:

```
karaf@root()> log:display -p "%d - %c - %m%n"
2015-07-01 07:01:58,007 - org.apache.sshd.common.util.SecurityUtils - BouncyCastle not
registered, using the default JCE provider
2015-07-01 07:01:58,725 - org.apache.aries.jmx.core - Starting JMX OSGi agent
2015-07-01 07:01:58,744 - org.apache.aries.jmx.core - Registering MBean with ObjectName
[osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framework,uuid=6361fc65
-8df4-4886-b0a6-479df2d61c83] for service with service.id [13]
2015-07-01 07:01:58,747 - org.apache.aries.jmx.core - Registering
org.osgi.jmx.service.cm.ConfigurationAdminMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framework,uuid=6361fc65-
8df4-4886-b0a6-479df2d61c83
```

The pattern is a regular Log4j pattern where you can use keywords like %d for the date, %c for the class, %m for the log message, etc.

`log:exception-display`

The `log:exception-display` command displays the last occurred exception.

As for `log:display` command, the `log:exception-display` command uses the `rootLogger` by default, but you can specify a logger with the `logger` argument.

`log:get`

The `log:get` command show the current log level of a logger.

By default, the log level showed is the one from the root logger:

```
karaf@root()> log:get
Logger | Level
--------------
ROOT   | INFO
```

You can specify a particular logger using the `logger` argument:

```
karaf@root()> log:get ssh
Logger | Level
--------------
ssh    | INFO
```

The `logger` argument accepts the `ALL` keyword to display the log level of all logger (as a list).

For instance, if you have defined your own logger in `etc/org.ops4j.pax.logging.cfg` file like this:

```
log4j.logger.my.logger = DEBUG
```

you can see the list of loggers with the corresponding log level:

```
karaf@root()> log:get ALL
Logger    | Level
-----------------
ROOT      | INFO
my.logger | DEBUG
```

The `log:list` command is an alias to `log:get ALL`.

`log:log`

The `log:log` command allows you to manually add a message in the log. It's interesting when you create Apache Karaf scripts:

```
karaf@root()> log:log "Hello World"
karaf@root()> log:display
2015-07-01 07:20:16,544 | INFO  | Local user karaf | command                          |
59 - org.apache.karaf.log.command - 4.0.0 | Hello World
```

By default, the log level is INFO, but you can specify a different log level using the `-l` option:

```
karaf@root()> log:log -l ERROR "Hello World"
karaf@root()> log:display
2015-07-01 07:21:38,902 | ERROR | Local user karaf | command                          |
59 - org.apache.karaf.log.command - 4.0.0 | Hello World
```

`log:set`

The `log:set` command sets the log level of a logger.

By default, it changes the log level of the `rootLogger`:

```
karaf@root()> log:set DEBUG
karaf@root()> log:get
Logger | Level
--------------
ROOT   | DEBUG
```

You can specify a particular logger using the `logger` argument, after the `level` one:

```
karaf@root()> log:set INFO my.logger
karaf@root()> log:get my.logger
Logger    | Level
-----------------
my.logger | INFO
```

The `level` argument accepts any Log4j log level: TRACE, DEBUG, INFO, WARN, ERROR, FATAL.

By it also accepts the DEFAULT special keyword.

The purpose of the DEFAULT keyword is to delete the current level of the logger (and only the level, the other properties like appender are not deleted) in order to use the level of the logger parent (loggers are hierarchical).

For instance, you have defined the following loggers (in `etc/org.ops4j.pax.logging.cfg` file):

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=DEBUG,appender2
```

You can change the level of `my.logger.custom` logger:

```
karaf@root()> log:set INFO my.logger.custom
```

Now we have:

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=INFO,appender2
```

You can use the DEFAULT keyword on `my.logger.custom` logger to remove the level:

```
karaf@root()> log:set DEFAULT my.logger.custom
```

Now we have:

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=appender2
```

It means that, at runtime, the `my.logger.custom` logger uses the level of its parent `my.logger`, so `INFO`.

Now, if we use the DEFAULT keyword with the `my.logger` logger:

```
karaf@root()> log:set DEFAULT my.logger
```

We have:

```
rootLogger=INFO,out,osgi:*
my.logger=appender1
my.logger.custom=appender2
```

So, both `my.logger.custom` and `my.logger` use the log level of the parent `rootLogger`.

It's not possible to use the DEFAULT keyword with the `rootLogger` as it doesn't have a parent.

`log:tail`

The `log:tail` is exactly the same as `log:display` but it continuously displays the log entries.

You can use the same options and arguments as for the `log:display` command.

By default, it displays the entries from the `rootLogger`:

```
karaf@root()> log:tail
2015-07-01 07:40:28,152 | INFO  | FelixStartLevel  | SecurityUtils                    |
16 - org.apache.sshd.core - 0.9.0 | BouncyCastle not registered, using the default JCE
provider
2015-07-01 07:40:28,909 | INFO  | FelixStartLevel  | core                             |
68 - org.apache.aries.jmx.core - 1.1.1 | Starting JMX OSGi agent
2015-07-01 07:40:28,928 | INFO  | FelixStartLevel  | core                             |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering MBean with ObjectName
[osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framework,uuid=b44a44b7
-41cd-498f-936d-3b12d7aafa7b] for service with service.id [13]
2015-07-01 07:40:28,936 | INFO  | JMX OSGi Agent   | core                             |
68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.service.cm.ConfigurationAdminMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framework,uuid=b44a44b7-
41cd-498f-936d-3b12d7aafa7b
```

To exit from the `log:tail` command, just type CTRL-C.

4.7.5. JMX LogMBean

All actions that you can perform with the `log:*` command can be performed using the LogMBean.

The LogMBean object name is `org.apache.karaf:type=log,name=*`.

Attributes

- `Level` attribute is the level of the ROOT logger.

Operations

- `getLevel(logger)` to get the log level of a specific logger. As this operation supports the ALL keyword, it returns a Map with the level of each logger.
- `setLevel(level, logger)` to set the log level of a specific logger. This operation supports the DEFAULT keyword as for the `log:set` command.

4.7.6. Advanced configuration

Filters

You can use filters on an appender. Filters allow log events to be evaluated to determine if or how they should be published.

Log4j provides ready to use filters:

- The DenyAllFilter (`org.apache.log4j.varia.DenyAllFilter`) drops all logging events. You can add this filter to the end of a filter chain to switch from the default "accept all unless instructed otherwise" filtering behaviour to a "deny all unless instructed otherwise" behaviour.
- The LevelMatchFilter (`org.apache.log4j.varia.LevelMatchFilter` is a very simple filter based on level matching. The filter admits two options `LevelToMatch` and `AcceptOnMatch`. If there is an exact match between the value of the `LevelToMatch` option and the level of the logging event, then the event is accepted in case the `AcceptOnMatch` option value is set to `true`. Else, if the `AcceptOnMatch` option value is set to `false`, the log event is rejected.
- The LevelRangeFilter (`org.apache.log4j.varia.LevelRangeFilter` is a very simple filter based on level matching, which can be used to reject messages with priorities outside a certain range. The filter admits three options `LevelMin`, `LevelMax` and `AcceptOnMatch`. If the log event level is between `LevelMin` and `LevelMax`, the log event is accepted if `AcceptOnMatch` is true, or rejected if `AcceptOnMatch` is false.
- The StringMatchFilter (`org.apache.log4j.varia.StringMatchFilter`) is a very simple filter based on string matching. The filter admits two options `StringToMatch` and `AcceptOnMatch`. If there is a match between the `StringToMatch` and the log event message, the log event is accepted if `AcceptOnMatch` is true, or rejected if `AcceptOnMatch` is false.

The filter is defined directly on the appender, in the `etc/org.ops4j.pax.logging.cfg` configuration file.

The format to use it:

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

For instance, you can use the `f1` LevelRangeFilter on the `out` default appender:

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

Thanks to this filter, the log files generated by the `out` appender will contain only log messages with a level between DEBUG and FATAL (the log events with TRACE as level are rejected).

Nested appenders

A nested appender is a special kind of appender that you use "inside" another appender. It allows you to create some kind of "routing" between a chain of appenders.

The most used "nested compliant" appender are:

- The AsyncAppender (`org.apache.log4j.AsyncAppender`) logs events asynchronously. This appender collects the events and dispatch them to all the appenders that are attached to it.
- The RewriteAppender (`org.apache.log4j.rewrite.RewriteAppender`) forwards log events to another appender after possibly rewriting the log event.

This kind of appender accepts an `appenders` property in the appender definition:

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

For instance, you can create a AsyncAppender named `async` and asynchronously dispatch the log events to a JMS appender:

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

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

Error handlers

Sometime, appenders can fail. For instance, a RollingFileAppender tries to write to the filesystem but the filesystem is full, or a JMS appender tries to send a message but the JMS broker is not there.

As logs can be very critical to you, you have to be informed that the log appender failed.

This is the purpose of the error handlers. Appenders may delegate their error handling to error handlers, giving a chance to react to the errors of the appender.

You have two error handlers available:

- The OnlyOnceErrorHandler (`org.apache.log4j.helpers.OnlyOnceErrorHandler`) implements log4j's default error handling policy which consists of emitting a message for the first error in an appender and ignoring all following errors. The error message is printed on `System.err`. This policy aims at protecting an otherwise working application from being flooded with error messages when logging fails.
- The FallbackErrorHandler (`org.apache.log4j.varia.FallbackErrorHandler`) allows a secondary appender to take over if the primary appender fails. The error message is printed on `System.err`, and logged in the secondary appender.

You can define the error handler that you want to use for each appender using the `errorhandler` property on the appender definition itself:

```
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

The `sift` appender is a OSGi oriented appender allowing you to split the log events based on MDC (Mapped Diagnostic Context) attributes.

MDC allows you to distinguish the different source of log events.

The `sift` appender provides OSGi oritend MDC attributes by default:

- `bundle.id` is the bundle ID
- `bundle.name` is the bundle symbolic name
- `bundle.version` is the bundle version

You can use these MDC properties to create a log file per bundle:

```
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=${karaf.log}/$\\{bundle.name\\}.log
log4j.appender.sift.appender.append=true
```

Enhanced OSGi stack trace renderer

By default, Apache Karaf provides a special stack trace renderer, adding some OSGi specific specific information.

In the stack trace, in addition of the class throwing the exception, you can find a pattern `[id:name:version]` at the end of each stack trace line, where:

- `id` is the bundle ID
- `name` is the bundle name
- `version` is the bundle version

It's very helpful to diagnosing the source of an issue.

For instance, in the following IllegalArgumentException stack trace, we can see the OSGi details about the source of the exception:

```
java.lang.IllegalArgumentException: Command not found:  *:foo
    at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:225)[21:org.apache.karaf.
shell.console:4.0.0]
    at
org.apache.felix.gogo.runtime.shell.Closure.executeStatement(Closure.java:162)[21:org.apac
he.karaf.shell.console:4.0.0]
    at
org.apache.felix.gogo.runtime.shell.Pipe.run(Pipe.java:101)[21:org.apache.karaf.shell.cons
ole:4.0.0]
    at
org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:79)[21:org.apache.karaf.s
hell.console:4.0.0]
    at
org.apache.felix.gogo.runtime.shell.CommandSessionImpl.execute(CommandSessionImpl.java:71)
[21:org.apache.karaf.shell.console:4.0.0]
    at
org.apache.karaf.shell.console.jline.Console.run(Console.java:169)[21:org.apache.karaf.she
ll.console:4.0.0]
    at java.lang.Thread.run(Thread.java:637)[:1.7.0_21]
```

Custom appenders

You can use your own appenders in Apache Karaf.

The easiest way to do that is to package your appender as an OSGi bundle and attach it as a fragment of the `org.ops4j.pax.logging.pax-logging-service` bundle.

For instance, you create `MyAppender`:

```
public class MyAppender extends AppenderSkeleton {
...
}
```

You compile and package as an OSGi bundle containing a MANIFEST looking like:

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

Copy your bundle in the Apache Karaf `system` folder. The `system` folder uses a standard Maven directory layout: groupId/artifactId/version.

In the `etc/startup.properties` configuration file, you define your bundle in the list before the pax-logging-service bundle.

You have to restart Apache Karaf with a clean run (purging the `data` folder) in order to reload the system bundles. You can now use your appender directly in `etc/org.ops4j.pax.logging.cfg` configuration file.

4.8. Configuration

4.8.1. Environment Variables & System Properties

Apache Karaf uses default configuration properties location in `etc` folder (`KARAF_ETC`).

You can override any configuration using either environment variables or system properties.

Basically, Apache Karaf is looking for property in env and system properties.

You can use `PID_PROPERTY=VALUE` syntax for environment variable and `pid.property=value` for system properties.

For instance, the following statement will override value `sshPort` in `etc/org.apache.karaf.shell.cfg`:

```
# environment variable
export ORG_APACHE_KARAF_SHELL_SSHPORT=8102
```

```
# system property
-Dorg.apache.karaf.shell.sshPort=8102
```

You can override any configuration property using this mechanism.

It's possible to "append" a new value based on the content of a configuration property. For instance, you want to add a new features repository to the default value (define the `etc/org.apache.karaf.features.cfg` config file `featuresRepositories` property. You can use the following env variable:

```
export
ORG_APACHE_KARAF_FEATURES_FEATURESREPOSITORIES='${featuresRepositories},mvn:org.apache.kar
af.decanter/apache-karaf-decanter/2.5.0/xml/features'
```

It means that Karaf will take value of `featuresRepositories` property contained in the `etc/org.apache.karaf.features.cfg` configuration file and add `mvn:org.apache.karaf.decanter/apache-karaf-decanter/2.5.0/xml/features`.

4.8.2. Files

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

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

Each configuration is identified by a ID (the ConfigAdmin PID). The configuration file names follow the `pid.cfg` name convention.

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

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

```
property=value
```

Properties can be referenced inside configuration files using the syntax `${<name>}`. Default and alternate values can be specified using `${<name>:-<default_value>}` and `${<name>:+<alternate_value>}` syntaxes respectively.

```
existing_property=baz
property1=${missing_property:-foo}   # "foo"
property2=${missing_property:+foo}   # empty string
property3=${existing_property:-bar}  # "baz"
property4=${existing_property:+bar}  # "bar"
```

Apache Karaf will look for system property as well. For instance, the following:

```
key=${foo}
```

means that `key` will contain `bar` if you start Karaf with `-Dfoo=bar`.

Environment variables can be referenced inside configuration files using the syntax `${env:<name>}` (e.g. `property=${env:FOO}` will set "property" to the value of the enviroment variable "FOO"). Default and alternate values can be defined for them as well using the same syntax as above.

Apache Karaf also supports other syntax, supporting:

- files on disk, for example to be used with Kubernetes secrets
- environment variables
- properties defined in `etc/config.properties`
- system properties

Secret Files

Usually secrets (for example when provided by Kubernetes) will surface as files in a location. By default, the location is `etc` folder. However, you can point to any folder. The configuration is located in `etc/config.properties`:

```
org.apache.felix.configadmin.plugin.interpolation.secretsdir=${karaf.etc}
```

The file contents are opaque and contain the secret value as-is. To use content of a secret file in a configuration property, you can do:

```
user=my-user
password=$[secret:db.password]
```

The content of `etc/db.password` will by used as value for `password` property.

Environment variables

Exactly the same as `${env:USER}`, you can use `$[env:USER]` syntax in a property:

```
greeting="Hello $[env:USER]"
```

With this syntax, default value is passed like this:

```
greeting="Hello $[env:USER;default=John]"
```

It's also possible to define the type of the property:

```
port=$[env:PORT;type=Integer]
```

System properties

System property can be passed with `-Dfoo=bar` to the Karaf process or define in `etc/config.properties`.

You can use these properties using the following syntax:

```
port=$[prop:my.port;default=8080]
```

cfg files

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

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

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

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

- `felix.fileinstall.enableConfigSave` flush back in the configuration file the changes performed directly on the configuration service (ConfigAdmin). If `true`, any change (using `config:*` commands, MBeans, OSGi service) is persisted back in the configuration false. Default is `true`.
- `felix.fileinstall.dir` is the directory where Apache Karaf is looking for configuration files. Default is `${karaf.etc}` meaning the value of the `KARAF_ETC` variable.

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

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

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

Apache Karaf persists configuration using its own persistence manager in the case of when available persistence managers do not support that. Configuration files are placed by default in `KARAF_ETC`, but it could be overridden via variable `storage` in `etc/org.apache.karaf.config.cfg`. If you want to disable the Karaf persistence manager, set the storage variable to an empty string (`storage=`).

4.8.3. `config:*` commands

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

`config:list`

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

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

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

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

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

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

`config:edit`

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

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

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

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

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

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

`config:property-list`

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

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

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

`config:property-set`

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

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

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

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

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

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

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

`config:property-append`

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

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

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

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

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

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

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

`config:property-delete`

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

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

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

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

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

`config:update` and `config:cancel`

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

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

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

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

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

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

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

`config:delete`

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

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

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

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

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

`config:meta`

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

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

```
karaf@root()> config:meta -p org.apache.karaf.log
Meta type informations for pid: org.apache.karaf.log
key     | name    | type   |
default                                                                    | description
-------------------------------------------------------------------------------------------
-------------------------------------------
size    | Size    | int    |
500                                                                        | size of the log to
keep in memory
pattern | Pattern | String | %d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} | %-32.32C
%4L | %m%n | Pattern used to display log entries
```

4.8.4. JMX ConfigMBean

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

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

Attributes

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

Operations

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

4.9. Artifacts repositories and URLs

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

```
<bundle>https://repo1.maven.org/maven2/org/apache/servicemix/nmr/org.apache.servicemix.nmr.api/1.0
```

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

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

You can deploy bundles from file system without using Maven

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

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

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

4.9.1. Maven URL Handler

The equivalent of the above bundle would be:

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

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

The `org.ops4j.pax.url.mvn` bundle resolves `mvn` URLs. It can be configured using the file `etc/org.ops4j.pax.url.mvn.cfg`. Full reference of `org.ops4j.pax.url.mvn` PID configuration can be found on the pax-web Wiki page (https://ops4j1.jira.com/wiki/display/paxurl/Aether+Configuration).

The most important property is:

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

Two other significant properties are:

- `org.ops4j.pax.url.mvn.defaultRepositories` : Comma separated list of locations that are checked before querying remote repositories. These can be treated as read-only repositories, as nothing is written there during artifact resolution.
- `org.ops4j.pax.url.mvn.localRepository` : by default (implicitly) it's the standard `~/.m2/repository` location. This local repository is used to store artifacts downloaded from one of remote repositories, so at next resolution attempt no remote request is issued.

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

```
org.ops4j.pax.url.mvn.repositories = http://www.example.org/repo@snapshots
```

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

4.9.2. Maven configuration commands

Full configuration of `org.ops4j.pax.url.mvn` bundle can be done using `org.ops4j.pax.url.mvn` PID (see `etc/org.ops4j.pax.url.mvn.cfg` file). This however may be cumbersome in some scenarios.

In order to make the user's life easier and to provide more *domain* oriented approach, Karaf provides several shell commands that makes Maven configuration easier.

maven:summary

This command shows a quick summary about current `org.ops4j.pax.url.mvn` PID configuration. For example:

```
karaf@root()> maven:summary -s

Option                     | Value                                                          | Sourc
---------------------------|----------------------------------------------------------------|------
Local repository           | /home/ggrzybek/.m2/repository                                  | Impli
Settings file              | /home/ggrzybek/.m2/settings.xml                                | Impli
Security settings file     | /home/ggrzybek/.m2/settings-security.xml                       | Impli
Global update policy       |                                                                | Impli
Global checksum policy     | warn                                                           | Defau
Update releases            | false                                                          | Defau
Require Config Admin       | true                                                           | Bundl
Use fallback repository    | false                                                          | Expli
Offline mode               | false                                                          | Defau
SSL/TLS certificate check  | true                                                           | Expli
Remote repositories        | https://repo1.maven.org/maven2/                                | PID c
                           | https://repository.apache.org/content/groups/snapshots-group/  | PID c
                           | https://oss.sonatype.org/content/repositories/snapshots/       | PID c
                           | https://oss.sonatype.org/content/repositories/ops4j-snapshots/ | PID c
Default repositories       | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/system/         | PID c
                           | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/kar/       | PID c
HTTP proxies               | proxy.everfree.forest:3128                                     | Maven
```

- `-s` option show where the value of the option come from. It may be implicit, explicit or default. We can also see whether the value was configured in PID or in `settings.xml` file.
- `-p` option uses original option names from `org.ops4j.pax.url.mvn` PID instead of descriptive option names
- `-d` option shows additional description, explaining what given option should be used for
- `-x` option turns on password display - if there's master password configured, it'll be displayed in clear text. This option may be used only by user with `admin` role.

maven:repository-list

This command displays all configured Maven repositories - in a much more readable way than the plain `config:proplist --pid org.ops4j.pax.url.mvn` command does.

```
karaf@root()> maven:repository-list -v

== Remote repositories
ID                              | URL
--------------------------------+-------------------------------------------------------------------------------
central                         | https://repo1.maven.org/maven2/
apache                          | https://repository.apache.org/content/groups/snapshots-group/
sonatype.snapshots.deploy       | https://oss.sonatype.org/content/repositories/snapshots/
ops4j.sonatype.snapshots.deploy | https://oss.sonatype.org/content/repositories/ops4j-snapshots/
special                         | https://repository.everfree.forest/

== Default repositories
ID                      | URL                                                          | Releases
------------------------+--------------------------------------------------------------+-------------
system.repository       | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/system/       | yes (daily)
kar.repository          | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/kar/     | yes (daily)
child.system.repository | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/system/       | yes (daily)
```

- `-v` option shows additional information about policies related to a given repository
- `-x` shows credentials for a given repository (if defined)

maven:password

`org.ops4j.pax.url.mvn` bundle uses the Aether library to handle Maven resolution. It uses the `settings.xml` file if credentials have to be used when accessing remote Maven repositories. This isn't done by `org.ops4j.pax.url.mvn`, but by Aether itself (or rather maven-settings library). When dealing with the `settings.xml` file, passwords that are stored there may need to be decrypted. Outside of Karaf, we can use `mvn -emp` and `mvn -ep` passwords and manually configure `~/.m2/settings-security.xml` file.

Karaf makes the task of managing credentials easier.

In order to use encrypted repository (or http proxy) passwords inside `settings.xml` file, Maven must know the *master password* stored inside `settings-security.xml` file. This file isn't usually present inside `~/.m2` directory and if there's a need to use it, one has to be created manually.

Here's the way to encrypt the Maven *master password* (which is used to encrypt ordinary passwords for repository or http proxies):

```
karaf@root()> maven:password -emp
Master password to encrypt: *****
Encrypted master password: {y+p9TiYuwVEHMHV14ej0Ni34zBnXXQrIOqjww/3Ro6U=}
```

The above usage simply prints the encrypted *master password*. We can however make this password persistent. This will result in the creation of a new `settings-security.xml` file and a change in the `org.ops4j.pax.url.mvn.security` property.

> **NOTE** Karaf maven commands will never overwrite your current `~/.m2/settings.xml` or `~/.m2/settings-security.xml` files. If there's a need to change these files, maven commands will make a copy of the existing file and set relevant `org.ops4j.pax.url.mvn` PID options to point to new locations.

```
karaf@root()> maven:password -emp --persist
Maven security settings will be stored in new file. This file will be used in org.ops4j.pax.url.mv
Master password to encrypt: *****
Encrypted master password: {lPPIFSUcPrMHnhwdauttAJYZcOe1D9sYGj4rwoaTwnY=}
New security settings stored in "/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/dat
karaf@root()> maven:summary -x

Option                     | Value
---------------------------+-------------------------------------------------------------------------
...
Security settings file     | /data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/mav
Master password            | admin
...
```

Now, when Maven *master password* is set, we can encrypt ordinary passwords that may be then used when defining/changing remote repositories or http proxies:

```
karaf@root()> maven:password -ep
Password to encrypt: *****
Encrypted password: {fHl8U3pINkEH7RR1CufRT+utj5gJHfqsRgd6wTo92Eo=}
You can use this encrypted password when defining repositories and proxies
```

Configuring repositories (default and remote)

As mentioned before, there are two kinds of repositories that are used/queried by `org.ops4j.pax.url.mvn` bundle when resolving `mvn:` based URIs:

default repositories

These are read-only local repositories that are simply queried before performing any remote access. The best example of such repository is `$KARAF_HOME/system` directory.

remote repositories

These are well-known Maven remote repositories - usually accessible over http(s) protocol. Popular repositories are Sonatype Nexus or JFrog Artifactory.

Both kinds of repositories may be created using the `maven:repository-add` command.

Here's how a default repository may be created:

```
karaf@root()> maven:repository-add --default -id my.default.repository --snapshots '${karaf.home}/

...
== Default repositories
ID                       | URL                                                                    | Re
-------------------------|------------------------------------------------------------------------|---
...
my.default.repository    | file:/data/servers/apache-karaf-4.2.0-SNAPSHOT/special-repository/ | yes
```

For a remote repository, we can specify more options (like credentials or update policies):

```
karaf@root()> maven:repository-add -idx 0 -id my.remote.repository --snapshots -up never --usernam
Maven settings will be updated and org.ops4j.pax.url.mvn.settings property will change. Continue?
New settings stored in "/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/maven-s

karaf@root()> maven:repository-list -x

== Remote repositories
ID                              | URL
--------------------------------|-------------------------------------------------------------------
my.remote.repository            | http://localhost/cloud-repository/
...
```

In the above example, a new `settings.xml` file was created. The reason is that although a new repository itself was added to `org.ops4j.pax.url.mvn.repositories` property, the credentials had to be stored in `settings.xml` file:

```
<servers>
  <server>
    <username>admin</username>
    <password>{fHl8U3pINkEH7RR1CufRT+utj5gJHfqsRgd6wTo92Eo=}</password>
    <id>my.remote.repository</id>
  </server>
</servers>
```

Here's summary of all options for `maven:repository-add` command:

- `-id` mandatory identifier of repository
- `-d` option may be used to configure default repositories instead of remote ones
- `-nr` option disables non-SNAPSHOT artifacts resolution in this repository
- `-s` option enables SNAPSHOT artifacts resolution in this repository
- `-up` sets *update policy* for given repository (`daily`, `always`, `never`, `interval:MINUTES`)
- `-cp` sets *checksum policy* for given repository (`fail`, `warn`, `ignore`)
- `-f` disables confirmation prompts for commands
- `-idx` allows to insert a repository at given position (instead of simply appending new repository at the end of current list of repositories)
- `-u` sets username for remote repository access
- `-p` sets password for remote repository access (may be encrypted using `maven:password -ep`)

After creating a repository, it may be deleted (using `maven:repository-remove` command) or changed (`maven:repository-change` command). All the options are the same as in `maven:repository-add` command. When removing a repository, only `-id` (and possibly `-d`) options are needed.

```
karaf@root()> repository-remove -d -id my.default.repository
Are you sure to remove repository with ID "my.default.repository" for URL file:/data/servers/apach

karaf@root()> repository-change -id special --username discord --password d1sc0rd
Maven settings will be updated and org.ops4j.pax.url.mvn.settings property will change. Continue?
New settings stored in "/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/maven-s
```

Configuring HTTP proxies

When accessing remote repositories using `org.ops4j.pax.url.mvn` (Aether library) there may be a need to let Maven/Aether know about HTTP proxies to use. HTTP proxies **can't be configured** inside `etc/org.ops4j.pax.url.mvn.cfg` file. It has to be done in `settings.xml` and its location has to be set in `org.ops4j.pax.url.mvn.settings` PID property.

`maven:http-proxy` command can be used to add/change/remove HTTP proxy definition. It automatically does a copy of the existing `settings.xml` file and changes `org.ops4j.pax.url.mvn.settings` PID property.

For example:

```
karaf@root()> maven:http-proxy-list -x

ID       | Host                   | Port | Non-proxy hosts | Username | Password
─────────┼────────────────────────┼──────┼─────────────────┼──────────┼─────────────
my.proxy | proxy.everfree.forest | 3128 | 192.168.2.*     | admin    | super-secret

karaf@root()> maven:http-proxy --remove -id my.proxy
New settings stored in "/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/maven-s

No HTTP proxies configured in /data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/m

karaf@root()> maven:http-proxy --add -id my.proxy --username discord --password '{fHl8U3pINkEH7RR1
New settings stored in "/data/servers/apache-karaf-4.2.0-SNAPSHOT/data/cache/bundle53/data/maven-s

karaf@root()> maven:http-proxy-list -x

ID       | Host                   | Port | Non-proxy hosts           | Username | Password
─────────┼────────────────────────┼──────┼───────────────────────────┼──────────┼─────────
my.proxy | proxy.everfree.forest | 3128 | 127.*|192.168.*|localhost | discord  | admin
```

Here's a summary of options for the `maven:http-proxy` command:

- `-id` identifier of HTTP proxy

- `-add` / `--change` / `--remove` is an operation to perform on proxy
- `-f` disables confirmation prompts for commands
- `-u` sets username for remote HTTP proxy
- `-p` sets password for remote HTTP proxy (may be encrypted using `maven:password -ep`)
- `-n` sets *non proxy hosts* option, which is `|`-separated list of glob patterns for IP addresses/ host names that should be accessed bypassing HTTP proxy

`maven:http-proxy` configures for example this section in `settings.xml`:

```
<proxies>
  <proxy>
    <username>discord</username>
    <password>{fHl8U3pINkEH7RR1CufRT+utj5gJHfqsRgd6wTo92Eo=}</password>
    <port>3128</port>
    <host>proxy.everfree.forest</host>
    <nonProxyHosts>127.*|192.168.*|localhost</nonProxyHosts>
    <id>my.proxy</id>
  </proxy>
</proxies>
```

4.10. Provisioning

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

4.10.1. Application

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

4.10.2. OSGi

It natively supports the deployment of OSGi applications.

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

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

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

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

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

4.10.3. Feature and resolver

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

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

A feature describes an application as:

- a name
- a version
- a optional description (eventually with a long description)
- a set of bundles
- optionally a set configurations or configuration files
- optionally a set of dependency features

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

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

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

```
serviceRequirements=default
```

The possible values are:

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

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

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

By default, the feature service is able to detect bundles which need to be refreshed. For instance, a bundle has to be refreshed:

1. when bundle A uses an import package, and bundle B is installed providing a new version of the package (matching bundle A import version range). Then bundle A has to be refreshed to use the new version of the package.
2. when bundle A uses an optional import package, and bundle B is installed providing this package. Then, bundle A has to be refreshed to actually use the package.
3. when bundle A uses a package provided by bundle B, then, bundle A will be refreshed as well. This is kind of "cascading" refresh.

Some users might be concerned about this refresh behavior, and prefer to manage refresh "by hand".

In that case, you can disable this auto refresh behavior using the `autoRefresh` property in `etc/org.apache.karaf.features.cfg` configuration file:

```
autoRefresh=false
```

By default, `autoRefresh` is `true`. Using `false` will disable auto refresh performed by the Karaf features service.

4.10.4. Features repositories

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

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

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

```
<features name="my-features-repo" xmlns="http://karaf.apache.org/xmlns/features/v1.4.0">
  <feature name="feature1" version="1.0.0">
    <bundle>...</bundle>
    <bundle>...</bundle>
  </feature>
  <feature name="feature2" version="1.1.0">
    <feature>feature1</feature>
    <bundle>...</bundle>
  </feature>
</features>
```

Here’s the same descriptor in JSON format:

```
{
 "name": "my-features-repo",
 "feature": [
    {
        "name": "feature1",
        "version": "1.0.0",
        "bundle": [
           { "location": "..." },
           { "location": "..."
        ]
    },
    {
        "name": "feature2",
        "version": "1.1.0",
        "feature": [
            { "name": "feature1" }
        ]
        "bundle": [
            { "location": "..." }
        ]
    }
 ]
}
```

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

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

The features state is stored in the Apache Karaf cache (in the `KARAF_DATA` folder). You can restart Apache Karaf, the previously installed features remain installed and available after restart. If you do a clean restart or you delete the Apache Karaf cache (delete the `KARAF_DATA` folder), all

previously registered features repositories and features installed will be lost: you will have to register the features repositories and install features by hand again. To prevent this behaviour, you can specify features as boot features.

4.10.5. Boot features

A boot feature is automatically installed by Apache Karaf, even if it has not been previously installed using `feature:install` or FeatureMBean.

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

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

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

To remove features from the `featuresBoot` list in the `etc/org.apache.karaf.features.cfg` configuration file:

1. Navigate to `etc/org.apache.karaf.features.cfg`.
2. Remove the undesired feature.
3. Restart your container.

> **NOTE** After the restart, the features will be present in the `etc/org.apache.karaf.features.cfg` configuration file, but they will not be installed and the undesired feature or behavior will no longer be present or active.

> **NOTE** Another way to clean up the `featuresBoot` is to stop Karaf, update `featuresBoot`, and remove the **data** folder.

4.10.6. Features upgrade

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

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

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

4.10.7. Overrides

Bundles defined in features can be overridden by using a file etc/overrides.properties. Each line in the file defines one override. The syntax is: <bundle-uri>[;range="[min,max)"] The given bundle will override all bundles in feature definitions with the same symbolic name if the version of the override is greater than the version of the overridden bundle and the range matches. If no range is given then compatibility on the micro version level is assumed.

So for example the override mvn:org.ops4j.pax.logging/pax-logging-service/1.8.5 would overide pax-logging-service 1.8.3 but not 1.8.6 or 1.7.0.

4.10.8. Feature bundles

Start Level

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

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

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

```
{
    "feature": [
        {
            "name": "my-project",
            "version": "1.0.0"
            "bundle": [
                {
                    "startLevel": 80,
                    "location": "mvn:com.mycompany.myproject/myproject-dao"
                },
                {
                    "startLeve": 85,
                    "location": "mvn:com.mycompany.myproject/myproject-service"
                }
            ]
        }
    ]
}
```

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

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

Simulate, Start and stop

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

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

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

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

```
{
    "feature": [
        {
            "name": "my-project",
            "version": "1.0.0"
            "bundle": [
                {
                    "startLevel": 80,
                    "start": false,
                    "location": "mvn:com.mycompany.myproject/myproject-dao"
                },
                {
                    "startLeve": 85,
                    "start": false,
                    "location": "mvn:com.mycompany.myproject/myproject-service"
                }
            ]
        }
    ]
}
```

Dependency

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

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

```
<bundle dependency="true">...</bundle>
```

```
{
    "dependency": true,
    "location": "..."
}
```

The dependency flag means that the bundle will be installed only if there's not other bundle providing the same capability.

4.10.9. Dependent features

A feature can depend on a set of other features:

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

```
{
    "feature": [
        {
            "name": "my-project",
            "version": "1.0.0",
            "feature": [
                { "name": "other" }
            ],
            "bundle": [
                {
                    "startLevel": 80,
                    "start": false,
                    "location": "mvn:com.mycompany.myproject/myproject-dao"
                },
                {
                    "startLevel": 85,
                    "start": false,
                    "location": "mvn:com.mycompany.myproject/myproject-service"
                }
            ]
        }
    ]
}
```

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

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

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

```
{
    "feature": [
        {
            "name": "spring-dm",
            "feature": [
                {
                    "name": "spring",
                    "version": "[2.5.6,4)"
                }
            ]
        }
    ]
}
```

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

If a single version is specified, the range will be considered open-ended.

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

To specify an exact version, use a closed range such as `[3.1,3.1]`.

Feature prerequisites

A prerequisite feature is a special kind of dependency. If you add the `prerequisite` attribute to dependant feature tag then it will force installation and also activation of bundles in the dependant feature before the installation of the actual feature. This may be handy in the case that bundles enlisted in a given feature are not using pre installed URLs such as `wrap` or `war`.

```
<feature name="foo" version="1.0.0">
    <feature prerequisite="true">bar</feature>
    ...
</feature>
```

```
{
    "name": "foo",
    "version": "1.0.0",
    "feature": [
        {
            "prerequisite": true,
            "name": "bar"
        }
    ]
}
```

4.10.10. Feature configurations

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

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

```
...
"config": [
    {
        "name": "com.foo.bar",
        "value": "myProperty=myValue"
    }
]
....
```

The `name` attribute of the `config` element corresponds to the configuration PID (see the [Configuration section|configuration] for details).

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

The content (value) of the `config` element is a set of properties, following the `key=value` standard.

4.10.11. Feature configuration files

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

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

```
  "configfile": [
    {
        "finalname": "/etc/myfile.cfg",
        "override": false,
        "location": "URL"
    }
  ]
```

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

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

For instance:

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

```
  "configfile": [
    {
        "finalname": "${karaf.etc}/myfile.cfg",
        "override": false,
        "location": "URL"
    }
  ]
```

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

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

Requirements

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

For instance, a feature can contain:

```
<requirement>osgi.ee;filter:=&amp;quot;(&amp;amp;(osgi.ee=JavaSE)(!(version&amp;gt;=1.8)))&amp;quot;</requ
irement>
```

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

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

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

4.10.12. Commands

`feature:repo-list`

The `feature:repo-list` command lists all registered feature repositories:

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

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

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

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

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

`feature:repo-add`

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

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

- a feature repository URL. It’s an URL directly to the features XML file. Any URL described in the [Artifacts repositories and URLs section|urls] of the user guide is supported.
- a feature repository name defined in the `etc/org.apache.karaf.features.repos.cfg` configuration file.

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

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

#
# This file describes the features repository URL
# It could be directly installed using feature:repo-add command
#
enterprise=mvn:org.apache.karaf.features/enterprise/LATEST/xml/features
spring=mvn:org.apache.karaf.features/spring/LATEST/xml/features
cellar=mvn:org.apache.karaf.cellar/apache-karaf-cellar/LATEST/xml/features
cave=mvn:org.apache.karaf.cave/apache-karaf-cave/LATEST/xml/features
camel=mvn:org.apache.camel.karaf/apache-camel/LATEST/xml/features
camel-extras=mvn:org.apache-extras.camel-extra.karaf/camel-extra/LATEST/xml/features
cxf=mvn:org.apache.cxf.karaf/apache-cxf/LATEST/xml/features
cxf-dosgi=mvn:org.apache.cxf.dosgi/cxf-dosgi/LATEST/xml/features
cxf-xkms=mvn:org.apache.cxf.services.xkms/cxf-services-xkms-features/LATEST/xml
activemq=mvn:org.apache.activemq/activemq-karaf/LATEST/xml/features
jclouds=mvn:org.apache.jclouds.karaf/jclouds-karaf/LATEST/xml/features
openejb=mvn:org.apache.openejb/openejb-feature/LATEST/xml/features
wicket=mvn:org.ops4j.pax.wicket/features/LATEST/xml/features
hawtio=mvn:io.hawt/hawtio-karaf/LATEST/xml/features
pax-cdi=mvn:org.ops4j.pax.cdi/pax-cdi-features/LATEST/xml/features
pax-jdbc=mvn:org.ops4j.pax.jdbc/pax-jdbc-features/LATEST/xml/features
pax-jpa=mvn:org.ops4j.pax.jpa/pax-jpa-features/LATEST/xml/features
pax-web=mvn:org.ops4j.pax.web/pax-web-features/LATEST/xml/features
pax-wicket=mvn:org.ops4j.pax.wicket/pax-wicket-features/LATEST/xml/features
ecf=http://download.eclipse.org/rt/ecf/latest/site.p2/karaf-features.xml
decanter=mvn:org.apache.karaf.decanter/apache-karaf-decanter/LATEST/xml/features
```

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

```
karaf@root()> feature:repo-add pax-jdbc
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/LATEST/xml/features
```

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

```
karaf@root()> feature:repo-add pax-jdbc 1.3.0
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features
```

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

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features
```

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

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

`feature:repo-refresh`

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

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

Without argument, the command refreshes all features repository:

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

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

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

```
karaf@root()> feature:repo-refresh pax-jdbc
Refreshing feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/LATEST/xml/features
```

`feature:repo-remove`

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

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

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

```
karaf@root()> feature:repo-remove org.ops4j.pax.jdbc-1.3.0
```

```
karaf@root()> feature:repo-remove mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/
features
```

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

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

```
karaf@root()> feature:repo-remove -u org.ops4j.pax.jdbc-1.3.0
```

`feature:list`

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

```
Name                          | Version                          | Required | State
| Repository               | Description
--------------------------------------------------------------------------------------
--------------------------------------------------------------------------
pax-cdi                        | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI support
pax-cdi-1.1                    | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support
pax-cdi-1.2                    | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support
pax-cdi-weld                   | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Weld CDI support
pax-cdi-1.1-weld               | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support
pax-cdi-1.2-weld               | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.2 support
pax-cdi-openwebbeans           | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | OpenWebBeans CDI support
pax-cdi-web                    | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Web CDI support
pax-cdi-1.1-web                | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support
...
```

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

```
karaf@root()> feature:list -o
Name                          | Version                          | Required | State
| Repository               | Description
--------------------------------------------------------------------------------------
--------------------------------------------------------------------------------
deltaspike-core               | 1.2.1                            |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike core support
deltaspike-data               | 1.2.1                            |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike data support
deltaspike-jpa                | 1.2.1                            |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike jpa support
deltaspike-partial-bean       | 1.2.1                            |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike partial bean support
pax-cdi                       | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI support
pax-cdi-1.1                   | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support
pax-cdi-1.1-web               | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support
pax-cdi-1.1-web-weld          | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.1 support
pax-cdi-1.1-weld              | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support
pax-cdi-1.2                   | 0.12.0                           |          | Uninstalled
| org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support
...
```

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

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

```
karaf@root()> feature:list -i
Name            | Version | Required | State   | Repository     | Description
------------------------------------------------------------------------------------------
------------------------
aries-proxy     | 4.0.0   |          | Started | standard-4.0.0 | Aries Proxy
aries-blueprint | 4.0.0   | x        | Started | standard-4.0.0 | Aries Blueprint
feature         | 4.0.0   | x        | Started | standard-4.0.0 | Features Support
shell           | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Shell
shell-compat    | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Shell
Compatibility
deployer        | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Deployer
bundle          | 4.0.0   | x        | Started | standard-4.0.0 | Provide Bundle support
config          | 4.0.0   | x        | Started | standard-4.0.0 | Provide OSGi
ConfigAdmin support
diagnostic      | 4.0.0   | x        | Started | standard-4.0.0 | Provide Diagnostic
support
instance        | 4.0.0   | x        | Started | standard-4.0.0 | Provide Instance support
jaas            | 4.0.0   | x        | Started | standard-4.0.0 | Provide JAAS support
log             | 4.0.0   | x        | Started | standard-4.0.0 | Provide Log support
package         | 4.0.0   | x        | Started | standard-4.0.0 | Package commands and
mbeans
service         | 4.0.0   | x        | Started | standard-4.0.0 | Provide Service support
system          | 4.0.0   | x        | Started | standard-4.0.0 | Provide System support
kar             | 4.0.0   | x        | Started | standard-4.0.0 | Provide KAR (KARaf
archive) support
ssh             | 4.0.0   | x        | Started | standard-4.0.0 | Provide a SSHd server
on Karaf
management      | 4.0.0   | x        | Started | standard-4.0.0 | Provide a JMX
MBeanServer and a set of MBeans in
wrap            | 0.0.0   | x        | Started | standard-4.0.0 | Wrap URL handler
```

`feature:install`

The `feature:install` command installs a feature.

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

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

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

```
karaf@root()> feature:install -t -v eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
  Managing bundle:
    org.apache.felix.metatype / 1.0.12
```

You can specify a feature version to install:

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

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

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

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

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

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

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

`feature:start`

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

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

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

`feature:stop`

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

`feature:uninstall`

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

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

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

4.10.13. Deployer

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

Apache Karaf provides a features deployer.

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

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

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

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

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

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

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

</features>
```

4.10.14. JMX FeatureMBean

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

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

Attributes

The FeatureMBean provides two attributes:

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

The `Repositories` attribute provides the following information:

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

The `Features` attribute provides the following information:

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

Operations

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

Notifications

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

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

- When a feature changes (installed or uninstalled).

4.11. Deployers

The following picture describes the architecture of the deployers.

Apache Karaf polls the `deploy` folder for new files.

You can configure the location of the `deploy` folder, and the polling behaviour in the `etc/org.apache.felix.fileinstall-deploy.cfg` configuration file:

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

felix.fileinstall.dir           = ${karaf.base}/deploy
felix.fileinstall.tmpdir        = ${karaf.data}/generated-bundles
felix.fileinstall.poll          = 1000
felix.fileinstall.start.level   = 80
felix.fileinstall.active.level  = 80
felix.fileinstall.log.level     = 3
```

- `felix.fileinstall.dir` defines the location of the `deploy` folder. Default value is `KARAF_BASE/deploy`.
- `felix.fileinstall.tmpdir` defines a temporary folder where the deployers store their files. Default value is `KARAF_DATA/generated-bundles`.
- `felix.fileinstall.poll` defines the polling interval (in milliseconds). Default value is 1 second.

When Apache Karaf polls a file from the `deploy` folder, it "delegates" the file handling to a deployer.

By default, Apache Karaf provides a set of deployers:

- Blueprint deployer is able to handle Blueprint XML files.
- Spring deployer is able to handle Spring XML files.
- Features deployer is able to handle Apache Karaf features XML files (see [Provisioning section|provisioning] for details).
- KAR deployer is able to handle KAR files (see [KAR section|kar] for details).
- Wrap deployer is able to handle non-OSGi jar files and turns it as OSGi bundles "on the fly".
- Optionally, WAR deployer (if you install the war feature) is able to handle WAR files.

4.11.1. Blueprint deployer

The Blueprint deployer is able to handle plain Blueprint XML configuration files.

The Blueprint deployer is able to transform "on the fly" any Blueprint XML file into a valid OSGi bundle.

The generated OSGi MANIFEST will contain the following headers:

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

The `name` and `version` of the file are extracted using a heuristic that will match common patterns.

For example `my-config-1.0.1.xml` will lead to `name = my-config` and `version = 1.0.1`.

The default imported packages are extracted from the blueprint file definition and includes all classes referenced directly.

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

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

4.11.2. Spring deployer

The Spring deployer is similar to the Blueprint deployer.

The Spring deployer is able to deploy Spring XML files.

Like the Blueprint deployer, the generated OSGi MANIFEST will contain the following headers:

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

If you need to customize the generated manifest, you can do so by including a XML element in your Spring configuration:

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

4.11.3. Features deployer

See the Provisioning section for details.

4.11.4. KAR deployer

See the KAR section for details.

4.11.5. War deployer

The installation of the WAR feature enables a WAR deployer.

It means that with the war feature installed, Apache Karaf is a complete OSGi WebContainer (like Tomcat) where you can deploy WAB (WebApplication Bundle) or pure WAR (WebApplication aRchive).

You can install the war feature with:

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

The WAR deployer supports:

- WAB files
- WAR files
- exploded WAR (as a directory named `*.war`).

The only requirement of the WAR deployer is that the archive contains the `WEB-INF/web.xml` file.

4.11.6. Wrap deployer

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

The wrap deployer creates "on the fly" an OSGi bundle with a non-OSGi jar file.

The wrap deployer looks for jar files in the deploy folder. A jar file 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 wrap deployer "transforms" non-OSGi jar file into an OSGi bundle.

The wrap 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
80 | Active   |  80 | 2.3                 | commons-lang
```

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()> bundle:headers 80

commons-lang (80)
-----------------
Specification-Title = Commons Lang
Tool = Bnd-2.1.0.20130426-122213
Specification-Version = 2.3
Specification-Vendor = Apache Software Foundation
Implementation-Version = 2.3
Generated-By-Ops4j-Pax-From = wrap:file:/opt/apache-karaf-4.0.0/deploy/commons-
lang-2.3.jar$Bundle-SymbolicName=commons-lang&Bundle-Version=2.3
Implementation-Vendor-Id = org.apache
Created-By = 1.7.0_21 (Oracle Corporation)
Implementation-Title = Commons Lang
Manifest-Version = 1.0
Bnd-LastModified = 1386339925753
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

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

You can specify some MANIFEST headers by specifying the headers as URL parameters.

In the URL parameters, you can specify the headers using the *$* character and *&* to separate the different headers. For instance:

```
karaf@root()> bundle:install -s 'wrap:mvn:jboss/jbossall-client/4.2.3.GA/$Bundle-
SymbolicName=jbossall-client&Bundle-Version=4.2.3.GA&Export-
Package=org.jboss.remoting;version="4.2.3.GA",\!*'
```

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

```
<bundle>wrap:mvn:jboss/jbossall-client/4.3.2.GA/$Bundle-SymbolicName=jbossall-
client&amp;Bundle-Version=4.3.2.GA&amp;Export-
Package=org.jboss.remoting;version=&quot;4.3.2.GA&quot;,!*</bundle>
```

4.12. KAR

As described in the Provisioning section, Apache Karaf features describe applications.

A feature defines different resources to resolve using URLs (for instance, bundles URLs, or configuration files URLs). As described in the [Artifacts repositories and URLs section|urls], Apache Karaf looks for artifacts (bundles, configuration files, ...) in the artifact repositories. Apache Karaf may have to download artifacts from remote repositories.

Apache Karaf provides a special type of artifact that package a features XML and all resources described in the features of this XML. This artifact is named a KAR (KAraf aRchive).

A KAR file is essentially a jar (so a zip file) which contains a set of feature descriptor and bundle jar files.

A KAR file contains a `repository` folder containing:

- a set of features XML files
- the artifacts following the Maven directory structure (`groupId/artifactId/version/artifactId-version.type`).

For instance, the `spring-4.0.0.kar` contains:

```
~$ unzip -l spring-4.0.0.kar
Archive:  spring-4.0.0.kar
  Length      Date    Time    Name
---------  ---------- -----   ----
      143  2013-12-06 10:52   META-INF/MANIFEST.MF
    12186  2013-12-06 10:52   repository/org/apache/karaf/features/spring/4.0.0/
spring-4.0.0-features.xml
   575389  2013-12-06 10:52   repository/commons-collections/commons-collections/3.2.1/
commons-collections-3.2.1.jar
   232019  2013-12-06 10:52   repository/commons-beanutils/commons-beanutils/1.8.3/
commons-beanutils-1.8.3.jar
   673109  2013-12-06 10:52   repository/org/apache/servicemix/bundles/
org.apache.servicemix.bundles.struts/1.3.10_1/
org.apache.servicemix.bundles.struts-1.3.10_1.jar
    37084  2013-12-06 10:52   repository/org/springframework/
org.springframework.web.struts/3.2.4.RELEASE/
org.springframework.web.struts-3.2.4.RELEASE.jar
     7411  2013-12-06 10:52   repository/org/springframework/
org.springframework.instrument/3.2.4.RELEASE/
org.springframework.instrument-3.2.4.RELEASE.jar
   246881  2013-12-06 10:52   repository/org/springframework/
org.springframework.transaction/3.2.4.RELEASE/
org.springframework.transaction-3.2.4.RELEASE.jar
    16513  2013-12-06 10:52   repository/org/apache/servicemix/bundles/
org.apache.servicemix.bundles.aopalliance/1.0_6/
org.apache.servicemix.bundles.aopalliance-1.0_6.jar
   881124  2013-12-06 10:52   repository/org/springframework/org.springframework.core/
3.2.4.RELEASE/org.springframework.core-3.2.4.RELEASE.jar
   199240  2013-12-06 10:52   repository/org/springframework/
org.springframework.expression/3.2.4.RELEASE/
org.springframework.expression-3.2.4.RELEASE.jar
   614646  2013-12-06 10:52   repository/org/springframework/org.springframework.beans/
3.2.4.RELEASE/org.springframework.beans-3.2.4.RELEASE.jar
   340841  2013-12-06 10:52   repository/org/springframework/org.springframework.aop/
3.2.4.RELEASE/org.springframework.aop-3.2.4.RELEASE.jar
   877369  2013-12-06 10:52   repository/org/springframework/org.springframework.context/
3.2.4.RELEASE/org.springframework.context-3.2.4.RELEASE.jar
   130224  2013-12-06 10:52   repository/org/springframework/
org.springframework.context.support/3.2.4.RELEASE/
org.springframework.context.support-3.2.4.RELEASE.jar
    30640  2013-12-06 10:52   repository/org/apache/karaf/deployer/
org.apache.karaf.deployer.spring/4.0.0/org.apache.karaf.deployer.spring-4.0.0.jar
    51951  2013-12-06 10:52   repository/org/springframework/org.springframework.aspects/
3.2.4.RELEASE/org.springframework.aspects-3.2.4.RELEASE.jar
   411175  2013-12-06 10:52   repository/org/springframework/org.springframework.jdbc/
3.2.4.RELEASE/org.springframework.jdbc-3.2.4.RELEASE.jar
    48049  2013-12-06 10:52   repository/javax/portlet/portlet-api/2.0/portlet-api-2.0.jar
   190883  2013-12-06 10:52   repository/org/springframework/
org.springframework.web.portlet/3.2.4.RELEASE/
```

```
org.springframework.web.portlet-3.2.4.RELEASE.jar
   635680  2013-12-06 10:52   repository/org/springframework/org.springframework.web/
3.2.4.RELEASE/org.springframework.web-3.2.4.RELEASE.jar
   645946  2013-12-06 10:52   repository/org/springframework/
org.springframework.web.servlet/3.2.4.RELEASE/
org.springframework.web.servlet-3.2.4.RELEASE.jar
   464911  2013-12-06 10:52   repository/org/springframework/org.springframework.test/
3.2.4.RELEASE/org.springframework.test-3.2.4.RELEASE.jar
    69784  2013-12-06 10:52   repository/org/springframework/osgi/spring-osgi-web/1.2.1/
spring-osgi-web-1.2.1.jar
    16030  2013-12-06 10:52   repository/org/apache/geronimo/specs/geronimo-jta_1.1_spec/
1.1.1/geronimo-jta_1.1_spec-1.1.1.jar
    32359  2013-12-06 10:52   repository/org/apache/geronimo/specs/geronimo-jms_1.1_spec/
1.1.1/geronimo-jms_1.1_spec-1.1.1.jar
   208684  2013-12-06 10:52   repository/org/springframework/org.springframework.jms/
3.2.4.RELEASE/org.springframework.jms-3.2.4.RELEASE.jar
    75672  2013-12-06 10:52   repository/org/springframework/org.springframework.oxm/
3.2.4.RELEASE/org.springframework.oxm-3.2.4.RELEASE.jar
   393607  2013-12-06 10:52   repository/org/springframework/org.springframework.orm/
3.2.4.RELEASE/org.springframework.orm-3.2.4.RELEASE.jar
   338559  2013-12-06 10:52   repository/org/apache/servicemix/bundles/
org.apache.servicemix.bundles.cglib/3.0_1/org.apache.servicemix.bundles.cglib-3.0_1.jar
    35859  2013-12-06 10:52   repository/org/springframework/osgi/spring-osgi-io/1.2.1/
spring-osgi-io-1.2.1.jar
   362889  2013-12-06 10:52   repository/org/springframework/osgi/spring-osgi-core/1.2.1/
spring-osgi-core-1.2.1.jar
   120822  2013-12-06 10:52   repository/org/springframework/osgi/spring-osgi-extender/
1.2.1/spring-osgi-extender-1.2.1.jar
    24231  2013-12-06 10:52   repository/org/springframework/osgi/spring-osgi-annotation/
1.2.1/spring-osgi-annotation-1.2.1.jar
    12597  2013-12-06 10:52   repository/org/apache/karaf/bundle/
org.apache.karaf.bundle.springstate/4.0.0/org.apache.karaf.bundle.springstate-4.0.0.jar
    31903  2013-12-06 10:52   repository/org/eclipse/gemini/blueprint/gemini-blueprint-io/
1.0.0.RELEASE/gemini-blueprint-io-1.0.0.RELEASE.jar
   578205  2013-12-06 10:52   repository/org/eclipse/gemini/blueprint/gemini-blueprint-
core/1.0.0.RELEASE/gemini-blueprint-core-1.0.0.RELEASE.jar
   178525  2013-12-06 10:52   repository/org/eclipse/gemini/blueprint/gemini-blueprint-
extender/1.0.0.RELEASE/gemini-blueprint-extender-1.0.0.RELEASE.jar
---------                     -------
  9803140                     38 files
```

As a KAR file is a simple zip file, you can create the KAR file by hand.

For instance, the following Unix commands create a very simple KAR file:

```
~$ mkdir repository
~$ cp /path/to/features.xml repository/features.xml
~$ cp /path/to/my.jar repository/my/project/my/1.0.0/my-1.0.0.jar
~$ zip -r my.kar repository
updating: repository/ (stored 0%)
  adding: repository/my/project/my/1.0.0/my-1.0.0.jar (deflated 0%)
```

You can create KAR files using Apache Maven, or directly in the Apache Karaf console.

4.12.1. Maven

Apache Karaf provides a Maven plugin: `karaf-maven-plugin`.

The Apache Karaf Maven plugin provides the `kar` goal.

The `kar` goal does:

1. Reads all features specified in the features XML.
2. For each feature described in the features XML, the goal resolves the bundles described in the feature.
3. The goal finally packages the features XML, and the resolved bundles in a zip file.

For instance, you can use the following POM to create `my-kar.kar`:

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

    <modelVersion>4.0.0</modelVersion>

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <version>4.0.0</version>
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </build>

</project>
```

To create the KAR file, simply type:

```
~$ mvn install
```

The kar will be installed in the `target` directory.

4.12.2. Commands

Apache Karaf provides `kar:*` commands to manage KAR archives.

`kar:list`

The `kar:list` command lists the installed KAR archives.

```
karaf@root()> kar:list
KAR Name
------------------
my-kar-1.0-SNAPSHOT
```

A KAR is identified by its name.

`kar:create`

Instead of using the `karaf-maven-plugin` or create the KAR archive by hand, you can use the `kar:create` command.

The `kar:create` command creates a KAR file using a registered features repository.

For instance, you want to create a KAR file for the Pax Web repository.

The `feature:repo-list` command gives you the list of registered features repositories:

```
karaf@root()> feature:repo-list
Repository                       | URL
---------------------------------------------------------------------------------------------
-------------
standard-4.0.0                   | mvn:org.apache.karaf.features/standard/4.0.0/xml/
features
enterprise-4.0.0                 | mvn:org.apache.karaf.features/enterprise/4.0.0/xml/
features
spring-4.0.0                     | mvn:org.apache.karaf.features/spring/4.0.0/xml/features
org.ops4j.pax.web-4.1.4          | mvn:org.ops4j.pax.web/pax-web-features/4.1.4/xml/
features
```

You can use one of these features repositories to create the kar file:

```
karaf@root()> kar:create org.ops4j.pax.web-4.1.4
Adding feature pax-jetty
Adding feature pax-http-whiteboard
Adding feature pax-war
Adding feature pax-http-tomcat
Adding feature pax-war-tomcat
Adding feature pax-http
Adding feature pax-http-jetty
Adding feature pax-jsf-support
Adding feature pax-jetty-spdy
Kar file created : /home/jbonofre/Downloads/apache-karaf-4.0.0/data/kar/
org.ops4j.pax.web-4.1.4.kar
```

You can see that the KAR file has been created in the `KARAF_DATA/kar` folder.

By default, the `kar:create` command creates a KAR file, packaging all features in the features descriptor.

You can provide the list of features that you want to package into the KAR file:

```
karaf@root()> kar:create org.ops4j.pax.web-4.1.4 pax-jetty pax-tomcat
Adding feature pax-jetty
Adding feature pax-tomcat
Kar file created : /opt/apache-karaf-4.1.4/data/kar/org.ops4j.pax.web-4.1.4.kar
```

`kar:install`

You can deploy a KAR file using `kar:install` command.

The `kar:install` command expects the KAR URL. Any URL described in the [Artifacts repositories and URLs section|urls] is supported by the `kar:install` command:

```
karaf@root()> kar:install file:/tmp/my-kar-1.0-SNAPSHOT.kar
```

The KAR file is uncompressed and used to populate the `KARAF_BASE/system` folder.

The Apache Karaf KAR service is looking for features XML files in the KAR file, registers the features XML and automatically installs all features described in the features repositories present in the KAR file.

Optionally, you can control if the bundles should be automatically started or not using `--no-start` option.

`kar:uninstall`

The `kar:uninstall` command uninstalls a KAR file (identified by a name).

By uninstall, it means that:

- the features previously installed by the KAR file are uninstalled
- delete (from the `KARAF_DATA/system` repository) all files previously "populated" by the KAR file

For instance, to uninstall the previously installed `my-kar-1.0-SNAPSHOT.kar` KAR file:

```
karaf@root()> kar:uninstall my-kar-1.0-SNAPSHOT
```

4.12.3. Deployer

Apache Karaf also provides a KAR deployer. It means that you can drop a KAR file directly in the `deploy` folder.

Apache Karaf will automatically install KAR files from the `deploy` folder.

You can change the behaviours of the KAR deployer in the `etc/org.apache.karaf.kar.cfg`:

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

#
# Enable or disable the refresh of the bundles when installing
# the features contained in a KAR file
#
noAutoRefreshBundles=false

#
# Enable or disable the automatic start of the bundles when installing
# the features contained in a KAR file
#
noAutoStartBundles=false

#
# Directory where the kar are stored (when downloaded from Maven for instance)
#
#karStorage=${karaf.data}/kar
```

By default, when the KAR deployer installs features, it refreshes the bundles already installed. You can disable the automatic bundles refresh by setting the `noAutoRefreshBundles` property to `false`.

4.12.4. JMX KarMBean

On the JMX layer, you have a MBean dedicated to the management of the KAR files.

The ObjectName to use is `org.apache.karaf:type=kar,name=*`.

Attributes

The `Kars` attributes provides the list of KAR files (name) installed.

Operations

- `install(url)` installs the KAR file at the given `url`.
- `install(url, noAutoStartBundles)` installs the KAR file at the given `url`, deciding if you want to automatically start the bundles or not.
- `create(repository, features)` creates a KAR file using the given features `repository` name, and optionally the list of `features` to include in the KAR file.
- `uninstall(name)` uninstalls a KAR file with the given `name`.

4.13. Instances

A instance is a complete new Apache Karaf runtime, isolated from the other ones.

The purpose is to easily create and manage a new Apache Karaf runtime without installing a complete distribution.

A instance is a new instance that you can launch separately from the root one, and deploy applications into. It means that each instance is run on a different JVM.

A instance does not contain a full copy of the Apache Karaf distribution, but only a set of the configuration files and data folder which contains all the runtime information, logs and temporary files.

4.13.1. Using the instance commands

The **instance** commands allow you to create and manage instances.

Creating instances

You create a new runtime instance by typing [`instance:create`|/commands/instance-create] in the Karaf console.

As shown in the following example, `instance: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()> instance:create test
```

The new instance is a fresh Apache Karaf instance. It uses the same default configuration files, as when you install a fresh Karaf distribution.

You can enable the verbose mode for the `instance:create` command using the `-v` option:

```
karaf@root()> instance:create -v test
Creating new instance on SSH port 8103 and registry port 1101 / RMI server port 44446 at:
/opt/karaf/instances/test
Creating dir: /opt/karaf/instances/test/bin
Creating dir: /opt/karaf/instances/test/etc
Creating dir: /opt/karaf/instances/test/system
Creating dir: /opt/karaf/instances/test/deploy
Creating dir: /opt/karaf/instances/test/data
Creating file: /opt/karaf/instances/test/etc/config.properties
Creating file: /opt/karaf/instances/test/etc/jre.properties
Creating file: /opt/karaf/instances/test/etc/custom.properties
Creating file: /opt/karaf/instances/test/etc/java.util.logging.properties
Creating file: /opt/karaf/instances/test/etc/org.apache.felix.fileinstall-deploy.cfg
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.features.obr.cfg
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.features.repos.cfg
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.log.cfg
Creating file: /opt/karaf/instances/test/etc/org.ops4j.pax.logging.cfg
Creating file: /opt/karaf/instances/test/etc/org.ops4j.pax.url.mvn.cfg
Creating file: /opt/karaf/instances/test/etc/users.properties
Creating file: /opt/karaf/instances/test/etc/keys.properties
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.features.cfg
Creating file: /opt/karaf/instances/test/etc/system.properties
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.shell.cfg
Creating file: /opt/karaf/instances/test/etc/org.apache.karaf.management.cfg
Creating file: /opt/karaf/instances/test/bin/karaf
Creating file: /opt/karaf/instances/test/bin/start
Creating file: /opt/karaf/instances/test/bin/stop
```

You can manually configure the different ports, the location of the instance, the Apache Karaf features URLs using different options of the `instance:create` command. You can have details about these options using the `--help` option.

Cloning an instance

Instead of creating a fresh instance, you can clone an existing instance using `instance:clone`.

The `instance:clone` command reuse the files from the source instance:

```
karaf@root()> instance:clone root test
```

You can have details about the cloning options using the `--help` option.

Changing the instance location

By default, the new instances storage is in the `KARAF_HOME/instance` directory. You find a directory with the name of the instance storing the different instance files.

You can change the location of the instance using the `-l` option to the `instance:create` and `instance:clone` commands:

```
karaf@root()> instance:create -v -l /tmp/test test
Creating new instance on SSH port 8102 and registry port 1100 / RMI server port 44445 at:
/tmp/test
Creating dir: /tmp/test/bin
Creating dir: /tmp/test/etc
Creating dir: /tmp/test/system
Creating dir: /tmp/test/deploy
Creating dir: /tmp/test/data
Creating file: /tmp/test/etc/config.properties
Creating file: /tmp/test/etc/jre.properties
Creating file: /tmp/test/etc/custom.properties
Creating file: /tmp/test/etc/java.util.logging.properties
Creating file: /tmp/test/etc/org.apache.felix.fileinstall-deploy.cfg
Creating file: /tmp/test/etc/org.apache.karaf.features.obr.cfg
Creating file: /tmp/test/etc/org.apache.karaf.features.repos.cfg
Creating file: /tmp/test/etc/org.apache.karaf.log.cfg
Creating file: /tmp/test/etc/org.ops4j.pax.logging.cfg
Creating file: /tmp/test/etc/org.ops4j.pax.url.mvn.cfg
Creating file: /tmp/test/etc/users.properties
Creating file: /tmp/test/etc/keys.properties
Creating file: /tmp/test/etc/org.apache.karaf.features.cfg
Creating file: /tmp/test/etc/system.properties
Creating file: /tmp/test/etc/org.apache.karaf.shell.cfg
Creating file: /tmp/test/etc/org.apache.karaf.management.cfg
Creating file: /tmp/test/bin/karaf
Creating file: /tmp/test/bin/start
Creating file: /tmp/test/bin/stop
```

Careful, it's not possible to change the location of an instance once it has been created.

NOTE

`instance:destroy` will remove the instance location for you. You don't have to remove the instance location "by hand".

Changing instance ports

You can change the SSH port number assigned to an instance using the `instance:ssh-port-change` command:

```
karaf@root()> instance:ssh-port-change test 8104
```

where test is the instance name and 8104 is the new SSH port number to use for the test instance.

You can change the RMI Registry port number (used by JMX) of an instance using the `instance:rmi-registry-port-change` command:

```
karaf@root()> instance:rmi-registry-port-change test 1102
```

where test is the instance name and 1102 is the new RMI Registry port number to use for the test instance.

You can also change the RMI Server port number (used by JMX too) of an instance using the `instance:rmi-server-port-change` command:

```
karaf@root()> instance:rmi-server-port-change test 44447
```

where test is the instance name and 44447 is the new RMI Server port number to use for the test instance.

> **NOTE** The instance has to be stopped to be able to change the port numbers.

Starting instances

New instances are created in a stopped state.

To start an instance, you can use the `instance:start` command:

```
karaf@root()> instance:start test
```

where test is the instance name.

Listing instances

To list the instances and their current status, you can use the `instance:list` command:

```
karaf@root()> instance:list
SSH Port | RMI Registry | RMI Server | State   | PID   | Name
------------------------------------------------------------
    8101 |         1099 |      44444 | Started | 19652 | root
    8104 |         1101 |      44446 | Stopped | 0     | test
```

An instance can be in the following status:

- Stopped: the instance is stopped.

- Starting: the instance is starting.
- Started: the instance is up and running. You can connect and use it.

Status of an instance

You can get directly the status of a given instance using the `instance:status` command:

```
karaf@root()> instance:status test
Started
```

where test is the instance name.

Connecting to an instance

You can connect to a running instance directly from the root one using the `instance:connect` command:

```
karaf@root()> instance:connect test
```

where *test* is the instance name where to connect to.

By default, this command will use the same username used on the root instance, and the password will be prompted.

You can use a different username using the `-u` or `--username` option. You can also provide the password using the `-p` or `--password` option.

If you don't provide any argument, you will logon on the instance:

```
karaf@test()>
```

Note the name of instance in the shell prompt (@test).

You can logoff from the instance and return back to the root instance using the `logout` command or CTRL-D key binding:

```
karaf@test()> logout
karaf@root()>
```

The `instance:connect` command accepts shell commands as argument. It allows you to directly execute commands or scripts on the instance:

```
karaf@root()> instance:connect test feature:list
Name                          | Version          | Installed | Repository                | Description
------------------------------------------------------------------------------------------------------------------------
standard                      | 4.0.0            | x         | standard-4.0.0            | Karaf standard feature
aries-annotation              | 4.0.0            |           | standard-4.0.0            | Aries Annotations
wrapper                       | 4.0.0            |           | standard-4.0.0            | Provide OS integration
service-wrapper               | 4.0.0            |           | standard-4.0.0            | Provide OS integration (alias to wrapper feature)
obr                           | 4.0.0            |           | standard-4.0.0            | Provide OSGi Bundle Repository (OBR) support
config                        | 4.0.0            | x         | standard-4.0.0            | Provide OSGi ConfigAdmin support
region                        | 4.0.0            | x         | standard-4.0.0            | Provide Region Support
...
```

Stop an instance

To stop an instance, you can connect to the instance (using `instance:connect`) and execute the `system:shutdown` command.

You can also use the `instance:stop` command:

```
karaf@root()> instance:stop test
```

where test is the instance name.

The instance will go to the "Stopped" state.

Destroy an instance

You can completely delete a stopped instance using the `instance:destroy` command:

```
karaf@root()> instance:destroy test
```

where test is the instance name.

NOTE

The `instance:destroy` deletes the instance store (the location where the instance files are stored).

Rename an instance

You can change the name of a stopped instance using the `instance:rename` command:

```
karaf@root()> instance:rename test newTest
```

where test is the current instance name, and newTest the new instance name.

4.13.2. Instance script

The `instance:*` commands require the root instance running.

But, you can also administrate directly instances without the root instance, using the `bin/instance` Unix script (or `bin/instance.bat` script on Windows).

You find the same actions that you can do with the `instance:*` commands in the `instance[.bat]` script:

```
bin/instance
Available commands:
  clone - Clones an existing container instance.
  create - Creates a new container instance.
  destroy - Destroys an existing container instance.
  list - Lists all existing container instances.
  opts-change - Changes the Java options of an existing container instance.
  rename - Rename an existing container instance.
  rmi-registry-port-change - Changes the RMI registry port (used by management layer) of
an existing container instance.
  rmi-server-port-change - Changes the RMI server port (used by management layer) of an
existing instance.
  ssh-port-change - Changes the secure shell port of an existing container instance.
  start - Start an existing container instance.
  status - Check the current status of an instance.
  stop - Stop an existing container instance.
Type 'command --help' for more help on the specified command.
```

For instance, to list all the instances, you can use the `instance` script with the `list` command:

```
bin/instance list
SSH Port | RMI Registry | RMI Server | State   | PID | Name
-----------------------------------------------------------
    8101 |         1099 |      44444 | Stopped | 0   | root
    8102 |         1100 |      44445 | Stopped | 0   | test
```

It's exactly the same as executing `instance:list` in the root instance.

You can obtain details about commands options and arguments using the `--help` option. For instance:

```
bin/instance rename --help
DESCRIPTION
        instance:rename

        Rename an existing container instance.

SYNTAX
        instance:rename [options] name new-name

ARGUMENTS
        name
                The name of the container instance to rename
        new-name
                The new name of the container instance

OPTIONS
        --help
                Display this help message
        -v, --verbose
                Display actions performed by the command (disabled by default)
```

4.13.3. JMX InstanceMBean

On the JMX layer, you have a MBean dedicated to the management of the instances: the InstanceMBean.

The ObjectName to use is `org.apache.karaf:type=instance,name=*`.

Attributes

The `Instances` attribute is a tabular data attribute providing details about the instances:

- `Is Root` (boolean): if true, the instance is the root instance, false else.

- `JavaOpts` (string): it contains the JVM arguments used by the instance.
- `Location` (string): it's the path to the instance storage.
- `Name` (string): it's the name of the instance.
- `Pid` (long): it's the current system process ID (PID) of the instance process.
- `RMI Registry Port` (int): it's the port number of the instance RMI Registry (JMX).
- `RMI Server Port` (int): it's the port number of the instance RMI Server (JMX).
- `SSH Port` (int): it's the port number of the instance SSH Server.
- `State` (string): it's the current status of the instance (Stopped, Starting, Started).

Operations

The InstanceMBean provides the following operations, corresponding to the previous `instance:*` commands:

- `createInstance(instanceName, sshPort, rmiRegistryPort, rmiServerPort, location, javaOpts, features, featuresUrls)`: create a new instance.
- `changeSshPort(instanceName, port)`: change the SSH port of an instance.
- `changeRmiServerPort(instanceName, port)`: change the RMI server port of an instance.
- `changeRmiRegistry(instanceName, port)`: change the RMI registry port of an instance.
- `changeJavaOpts(instanceName, javaOpts)`: change the Java options of an instance.
- `destroyInstance(instanceName)`: destroy an instance.
- `startInstance(instanceName)`: start an instance.
- `startInstance(instanceName, options)`: start an instance with the given Java options.
- `startInstance(instanceName, options, wait, debug)`: start an instance with the given Java options. If wait is true, this operation is waiting for the instance is in "Started" state. If debug is true, the instance is started in debug mode.
- `stopInstance(instanceName)`: stop an instance.
- `renameInstance(instanceName, newInstanceName)`: rename an instance.
- `renameInstance(instanceName, newInstanceName, verbose)`: rename an instance. If verbose is true, this operation provides details in the log.

- `cloneInstance(instanceName, cloneName, sshPort, rmiRegistryPort, rmiServerPort, location, javaOpts)` : clone an existing instance.

4.14. Security

Apache Karaf provides an advanced and flexible security system, powered by JAAS (Java Authentication and Authorization Service) in an OSGi compliant way.

It provides a dynamic security system.

The Apache Karaf security framework is used internally to control the access to:

- the OSGi services (described in the developer guide)
- the console commands
- the JMX layer
- the WebConsole

Your applications can also use the security framework (see the developer guide for details).

NOTE

For security reason, by default, `karaf` user is disabled. To allow the logon, you have to have an user. You can enable `karaf` user by uncommenting the user in `etc/users.properties`:

```
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

4.14.1. Realms

Apache Karaf is able to manage multiple realms. A realm contains the definition of the login modules to use for the authentication and/or authorization on this realm. The login modules define the authentication and authorization for the realm.

The `jaas:realm-list` command lists the current defined realms:

```
karaf@root()> jaas:realm-list
Index | Realm Name | Login Module Class Name
---------------------------------------------------------------------------------
1     | karaf      | org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     | karaf      | org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

You can see that the Apache Karaf provides a default realm named `karaf`.

This realm has two login modules:

- the `PropertiesLoginModule` uses the `etc/users.properties` file as backend for users, groups, roles and password. This login module authenticates the users and returns the users' roles.
- the `PublickeyLoginModule` is especially used by the SSHd. It uses the `etc/keys.properties` file. This file contains the users and a public key associated to each user.

Apache Karaf provides additional login modules (see the developer guide for details):

- JDBCLoginModule uses a database as backend
- LDAPLoginModule uses a LDAP server as backend
- SyncopeLoginModule uses Apache Syncope as backend
- OsgiConfigLoginModule uses a configuration as backend
- Krb5LoginModule uses a Kerberos Server as backend
- GSSAPILdapLoginModule uses a LDAP server as backend but delegate LDAP server authentication to an other backend (typically Krb5LoginModule)

You can manage an existing realm, login module, or create your own realm using the `jaas:realm-manage` command.

Adding Realms or Login Modules

If a new realm should be added Apache Karaf provides an easy way to create a new Realm (although with limited flexibility compared to other approaches like blueprint or directly via DS). The `jaas:realm-add` command can be used for that purpose. Note, that it takes at least 2 Parameters (name of the realm and class name of the initial Login Module) or even more, if the

Login Module needs parameters. For example, a new realm `myrealm` which uses the `PropertiesLoginModule` with a `users` file located in `/tmp/users` can be added by the command

```
jaas:realm-add myrealm org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
users "/tmp/users"
```

To add a new Login Module to an existing realm the `jaas:module-add` command can be used. It is similar in semantics than the `jaas:realm-add` command except that it takes no realm name. Instead, the realm to modify has to be selected via `jaas:realm-manage` previously. Note that the options for the Login Module are given as a list of parameters which is interpreted as a map in the order "key1 value1 key2 value2". So the command

```
jaas:module-add org.apache.karaf.jaas.modules.properties.PropertiesLoginModule users
"/tmp/users"
```

will add a `PropertiesLoginModule` with parameters map `users -> /tmp/users` to the selected realm.

4.14.2. Users, groups, roles, and passwords

As we saw, by default, Apache Karaf uses a PropertiesLoginModule.

This login module uses the `etc/users.properties` file as storage for the users, groups, roles and passwords.

The initial `etc/users.properties` file contains:

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

#
# This file contains the users, groups, and roles.
# Each line has to be of the format:
#
# USER=PASSWORD,ROLE1,ROLE2,...
# USER=PASSWORD,_g_:GROUP,...
# _g_\:GROUP=ROLE1,ROLE2,...
#
# All users, grousp, and roles entered in this file are available after Karaf startup
# and modifiable via the JAAS command group. These users reside in a JAAS domain
# with the name "karaf".
#
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,ssh
```

We can see in this file, that we have one user by default: `karaf`. The default password is `karaf`.

The `karaf` user is member of one group: the `admingroup`.

A group is always prefixed by `_g_:`. An entry without this prefix is a user.

A group defines a set of roles. By default, the `admingroup` defines `group`, `admin`, `manager`, and `viewer` roles.

It means that the `karaf` user will have the roles defined by the `admingroup`.

Commands

The `jaas:*` commands manage the realms, users, groups, roles in the console.

`jaas:realm-list`

We already used the `jaas:realm-list` previously in this section.

The `jaas:realm-list` command lists the realm and the login modules for each realm:

```
karaf@root()> jaas:realm-list
Index | Realm Name | Login Module Class Name
--------------------------------------------------------------------------------
1     | karaf      | org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     | karaf      | org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

We have here one realm (`karaf`) containing two login modules (`PropertiesLoginModule` and `PublickeyLoginModule`).

The `index` is used by the `jaas:realm-manage` command to easily identify the realm/login module that we want to manage.

`jaas:realm-manage`

The `jaas:realm-manage` command switch in realm/login module edit mode, where you can manage the users, groups, and roles in the login module.

To identify the realm and login module that you want to manage, you can use the `--index` option. The indexes are displayed by the `jaas:realm-list` command:

```
karaf@root()> jaas:realm-manage --index 1
```

Another way is to use the `--realm` and `--module` options. The `--realm` option expects the realm name, and the `--module` option expects the login module class name:

```
karaf@root()> jaas:realm-manage --realm karaf --module
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
```

`jaas:user-list`

When you are in edit mode, you can list the users in the login module using the `jaas:user-list`:

```
karaf@root()> jaas:user-list
User Name | Group      | Role
--------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
```

You can see the user name and the group by role.

`jaas:user-add`

The `jaas:user-add` command adds a new user (and the password) in the currently edited login module:

```
karaf@root()> jaas:user-add foo bar
```

To "commit" your change (here the user addition), you have to execute the `jaas:update` command:

```
karaf@root()> jaas:update
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name | Group      | Role
--------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
foo       |            |
```

On the other hand, if you want to rollback the user addition, you can use the `jaas:cancel` command.

`jaas:user-delete`

The `jaas:user-delete` command deletes a user from the currently edited login module:

```
karaf@root()> jaas:user-delete foo
```

Like for the `jaas:user-add` command, you have to use the `jaas:update` to commit your change (or `jaas:cancel` to rollback):

```
karaf@root()> jaas:update
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name | Group      | Role
--------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
```

`jaas:group-add`

The `jaas:group-add` command assigns a group (and eventually creates the group) to a user in the currently edited login module:

```
karaf@root()> jaas:group-add karaf mygroup
```

`jaas:group-delete`

The `jaas:group-delete` command removes a user from a group in the currently edited login module:

```
karaf@root()> jaas:group-delete karaf mygroup
```

`jaas:group-role-add`

The `jaas:group-role-add` command adds a role in a group in the currently edited login module:

```
karaf@root()> jaas:group-role-add mygroup myrole
```

`jaas:group-role-delete`

The `jaas:group-role-delete` command removes a role from a group in the currently edited login module:

```
karaf@root()> jaas:group-role-delete mygroup myrole
```

`jaas:update`

The `jaas:update` command commits your changes in the login module backend. For instance, in the case of the PropertiesLoginModule, the `etc/users.properties` will be updated only after the execution of the `jaas:update` command.

`jaas:cancel`

The `jaas:cancel` command rollback your changes and doesn't update the login module backend.

4.14.3. Passwords encryption

By default, the passwords are stored in clear form in the `etc/users.properties` file.

It's possible to enable encryption in the `etc/org.apache.karaf.jaas.cfg` configuration file:

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

#
# Boolean enabling / disabling encrypted passwords
#
encryption.enabled = false

#
# Encryption Service name
#   the default one is 'basic'. Note that this digests without salting.
#   a more powerful one named 'jasypt' is available when installing the
#   encryption feature
#
encryption.name = basic

#
# Encryption prefix
#
encryption.prefix = {CRYPT}

#
# Encryption suffix
#
encryption.suffix = {CRYPT}

#
# Set the encryption algorithm to use in Karaf JAAS login module
# Supported encryption algorithms follow:
#   MD2
#   MD5
#   SHA-1
#   SHA-256
```

```
#   SHA-384
#   SHA-512
#
encryption.algorithm = SHA-256

#
# Encoding of the encrypted password.
# Can be:
#   hexadecimal
#   base64
#
encryption.encoding = hexadecimal
```

If the `encryption.enabled` property is set to true, the password encryption is enabled.

With encryption enabled, the password are encrypted at the first time a user logs in. The encrypted passwords are prefixed and suffixed with `\{CRYPT\}`. To re-encrypt the password, you can reset the password in clear (in `etc/users.properties` file), without the `\{CRYPT\}` prefix and suffix. Apache Karaf will detect that this password is in clear (because it's not prefixed and suffixed with `\{CRYPT\}`) and encrypt it again.

The `etc/org.apache.karaf.jaas.cfg` configuration file allows you to define advanced encryption behaviours:

- the `encryption.prefix` property defines the prefix to "flag" a password as encrypted. The default is `\{CRYPT\}`.
- the `encryption.suffix` property defines the suffix to "flag" a password as encrypted. The default is `\{CRYPT\}`.
- the `encryption.algorithm` property defines the algorithm to use for encryption (digest). The possible values are `MD2`, `MD5`, `SHA-1`, `SHA-256`, `SHA-384`, `SHA-512`. The default is `SHA-256` since Karaf 4.3.0, prior to this it was `MD5`. When the `basic` algorithm is used then the value is digested using this algorithm without salting. For a more secure alternative which uses salts please consider using the `jasypt` encryption algorithm instead.
- the `encryption.encoding` property defines the encoding of the encrypted password. The possible values are `hexadecimal` or `base64`. The default value is `hexadecimal`.

4.14.4. Managing authentication by key

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

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

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

```
user=key,role
```

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

You can create a key pair using:

```
ssh-keygen -t rsa -f karaf.id_rsa
```

NB: you can provide a passphrase with `-N` option to `ssh-keygen` but, in that case, you will need to enter the passphrase to allow the SSH client to use the key.

You have now the public and private keys:

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

You copy the key from `karaf.id_rsa.pub` file in the `etc/keys.properties`:

```
karaf=AAAAB3NzaC1yc2EAAAADAQABAAABAQCtXN9ZZ+K67UFbxZMxHsoR69vHNbN8qi17v/
5jF83FUexRqu8FvWCInBoW7eVFyeIiFXGd/
zaCrDHrZsqpwXNEha3ifvfGTY2+gMLfZZFgh2LFubXBH6G725XKs9aus+KLrwC8u/
uPr9Sw3YeSb0zxrjiXqv6hGYhaAHnskAgRCUxa+P4/JYNVS/
2+ZrvBrVMAwgEuwt1Y1IDYsXQRmLJPn5ayMCfzPTANXfgB7Hix
72f2XpHV3FdnKTbYwdA32Bg4ptJkuvyMXnBy5y7ChRU15OYGRToC4ETcPF2DB0EPOcbOLsQlNTKKRYuNR1zEpp6RAf
iWD65kmYK766CE8AbB,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

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

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

or to ssh

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

4.14.5. RBAC

Apache Karaf uses the roles to control the access to the resources: it's a RBAC (Role Based Access Control) system.

The roles are used to control:

- access to OSGi services
- access to the console (control the execution of the commands)
- access to JMX (MBeans and/or operations)
- access to the WebConsole

OSGi services

The details about OSGi services RBAC support is explained in the developer guide.

Console

Console RBAC supports is a specialization of the OSGi service RBAC. Actually, in Apache Karaf, all console commands are defined as OSGi services.

The console command name follows the `scope:name` format.

The ACL (Access Lists) are defined in `etc/org.apache.karaf.command.acl.<scope>.cfg` configuration files, where `<scope>` is the commands scope.

For instance, we can define the ACL to the `feature:*` commands by creating a `etc/org.apache.karaf.command.acl.feature.cfg` configuration file. In this `etc/org.apache.karaf.command.acl.feature.cfg` configuration file, we can set:

```
list = viewer
info = viewer
install = admin
uninstall = admin
```

Here, we define that `feature:list` and `feature:info` commands can be executed by users with `viewer` role, whereas the `feature:install` and `feature:uninstall` commands can only be executed by users with `admin` role. Note that users in the admin group will also have viewer role, so will be able to do everything.

Apache Karaf command ACLs can control access using (inside a given command scope):

- the command name regex (e.g. `name = role`)
- the command name and options or arguments values regex (e.g. `name[/.*[0-9][0-9][0-9]+.*/] = role` to execute name only with argument value above 100)

Both command name and options/arguments support exact matching or regex matching.

By default, Apache Karaf defines the following commands ACLs:

- `etc/org.apache.karaf.command.acl.bundle.cfg` configuration file defines the ACL for `bundle:*` commands. This ACL limits the execution of `bundle:*` commands for system bundles only to the users with `admin` role, whereas `bundle:*` commands for non-system bundles can be executed by the users with `manager` role.
- `etc/org.apache.karaf.command.acl.config.cfg` configuration file defines the ACL for `config:*` commands. This ACL limits the execution of `config:*` commands with `jmx.acl.*`, `org.apache.karaf.command.acl.*`, and `org.apache.karaf.service.acl.*` configuration PID to the users with `admin` role. For the other configuration PID, the users with the `manager` role can execute `config:*` commands.
- `etc/org.apache.karaf.command.acl.feature.cfg` configuration file defines the ACL for `feature:*` commands. Only the users with `admin` role can execute `feature:install`, `feature:uninstall`, `feature:start`, `feature:stop` and `feature:update` commands. The other `feature:*` commands can be executed by any user.
- `etc/org.apache.karaf.command.acl.jaas.cfg` configuration file defines the ACL for `jaas:*` commands. Only the users with `admin` role can execute `jaas:update` command. The other `jaas:*` commands can be executed by any user.
- `etc/org.apache.karaf.command.acl.kar.cfg` configuration file defines the ACL for `kar:*` commands. Only the users with `admin` role can execute `kar:install` and `kar:uninstall` commands. The other `kar:*` commands can be executed by any user.

- `etc/org.apache.karaf.command.acl.shell.cfg` configuration file defines the ACL for `shell:*` and "direct" commands. Only the users with `admin` role can execute `shell:edit`, `shell:exec`, `shell:new`, and `shell:java` commands. The other `shell:*` commands can be executed by any user.
- `etc/org.apache.karaf.command.acl.system.cfg` configuration file defines the ACL for `system:*` commands. Only the users with `admin` role can execute `system:property` and `system:shutdown` commands. Users with `manager` role can call `system:start-level` above 100, otherwise `admin` role is required. Also users with `viewer` role can obtain the current start-level. The other `system:*` commands can be executed by any user.

You can change these default ACLs, and add your own ACLs for additional command scopes (for instance `etc/org.apache.karaf.command.acl.cluster.cfg` for Apache Karaf Cellar, `etc/org.apache.karaf.command.acl.camel.cfg` from Apache Camel, ...).

You can fine tune the command RBAC support by editing the `karaf.secured.services` property in `etc/system.properties`:

```
#
# By default, only Karaf shell commands are secured, but additional services can be
# secured by expanding this filter
#
karaf.secured.services = (&(osgi.command.scope=*)(osgi.command.function=*))
```

JMX

Like for the console commands, you can define ACL (AccessLists) to the JMX layer.

The JMX ACL are defined in `etc/jmx.acl<ObjectName>.cfg` configuration file, where `<ObjectName>` is a MBean object name (for instance `org.apache.karaf.bundle` represents `org.apache.karaf;type=Bundle` MBean).

The `etc/jmx.acl.cfg` is the most generic configuration file and is used when no specific ones are found. It contains the "global" ACL definition.

JMX ACLs can control access using (inside a JMX MBean):

- the operation name regex (e.g. `operation* = role`)
- the operation arguments value regex (e.g. `operation(java.lang.String, int)[/([1-4])?[0-9]/,/.*/] = role`)

By default, Apache Karaf defines the following JMX ACLs:

- `etc/jmx.acl.org.apache.karaf.bundle.cfg` configuration file defines the ACL for the `org.apache.karaf:type=bundle` MBean. This ACL limits the `setStartLevel()`, `start()`, `stop()`, and `update()` operations for system bundles for only users with `admin` role. The other operations can be performed by users with the `manager` role.
- `etc/jmx.acl.org.apache.karaf.config.cfg` configuration file defines the ACL for the `org.apache.karaf:type=config` MBean. This ACL limits the change on `jmx.acl*`, `org.apache.karaf.command.acl*`, and `org.apache.karaf.service.acl*` configuration PIDs for only users with `admin` role. The other operations can be performed by users with the `manager` role.
- `etc/jmx.acl.org.apache.karaf.security.jmx.cfg` configuration file defines the ACL for the `org.apache.karaf:type=security,area=jmx` MBean. This ACL limits the invocation of the `canInvoke()` operation for the users with `viewer` role.
- `etc/jmx.acl.osgi.compendium.cm.cfg` configuration file defines the ACL for the `osgi.compendium:type=cm` MBean. This ACL limits the changes on `jmx.acl*`, `org.apache.karaf.command.acl*`, and `org.apache.karaf.service.acl*` configuration PIDs for only users with `admin` role. The other operations can be performed by users with the `manager` role.
- `etc/jmx.acl.java.lang.Memory.cfg` configuration file defines the ACL for the core JVM Memory MBean. This ACL limits the invocation of the `gc` operation for only users with the `manager` role.
- `etc/jmx.acl.cfg` configuration file is the most generic file. The ACLs defined here are used when no other specific ACLs match (by specific ACL, it's an ACL defined in another MBean specific `etc/jmx.acl.*.cfg` configuration file). The `list*()`, `get*()`, `is*()` operations can be performed by users with the `viewer` role. The `set*()` and all other `*()` operations can be performed by users with the `admin` role.

WebConsole

The Apache Karaf WebConsole is not available by default. To enable it, you have to install the `webconsole` feature:

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

The WebConsole doesn't support fine grained RBAC like console or JMX for now.

All users with the `admin` role can logon to the WebConsole and perform any operations.

4.14.6. SecurityMBean

Apache Karaf provides a JMX MBean to check if the current user can invoke a given MBean and/or operation.

The `canInvoke()` operation gets the roles of the current user, and check if one the roles can invoke the MBean and/or the operation, eventually with a given argument value.

Operations

- `canInvoke(objectName)` returns `true` if the current user can invoke the MBean with the `objectName`, `false` else.
- `canInvoke(objectName, methodName)` returns `true` if the current user can invoke the operation `methodName` on the MBean with the `objectName`, `false` else.
- `canInvoke(objectName, methodName, argumentTypes)` returns `true` if the current user can invoke the operation `methodName` with the array of arguments types `argumentTypes` on the MBean with `objectName`, `false` else.
- `canInvoke(bulkQuery)` returns a tabular data containing for each operation in the `bulkQuery` tabular data if `canInvoke` is `true` or `false`.

4.14.7. Security providers

Some applications require specific security providers to be available, such as [BouncyCastle|http://www.bouncycastle.org].

The JVM imposes some restrictions about the use of such jars: they have to be signed and be available on the boot classpath.

One way to deploy those providers is to put them in the JRE folder at `$JAVA_HOME/jre/lib/ext` and modify the security policy configuration (`$JAVA_HOME/jre/lib/security/java.security`) in order to register such providers.

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

Apache Karaf offers a simple way to configure additional security providers:

- put your provider jar in `lib/ext`

- modify the `etc/config.properties` configuration file to add the following property

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

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

For instance, to add the bouncycastle security provider, you define:

```
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*
```

4.15. Docker

Apache Karaf provides Docker resources allowing you to easily create your own image and container.

Official Karaf docker image are also available on Docker Hub.

But, Apache Karaf also provides a docker feature that allows you to:

- manipulate Docker containers directly from Apache Karaf
- create a Docker container based on the current running Apache Karaf instance (named provisioning)

4.15.1. Docker images

You can find the docker resource at https://github.com/apache/karaf/tree/master/assemblies/docker.

As prerequisites, you have:

- to install the most recent stable version of docker (https://docs.docker.com/installation/)

- to install the most recent stable version of docker-compose (https://docs.docker.com/compose/install/)

Official images

> **NOTE** The official Docker Karaf images are in preparation, as part of the Karaf build process. They should be available soon.

Apache Karaf official docker images should be available soon on Docker HUB: https://hub.docker.com/_/karaf/

You can directly pull the official image:

```
docker pull karaf
```

Build your own

You can create your own docker image. The images are based on the official Java Alpine (OpenJDK 8) image. If you want to build the Karaf image run:

```
sh build.sh
```

or

```
docker build -t karaf .
```

If you want to build the container for a specific version of Karaf you can configure it with the KARAF_VERSION arg:

```
docker build --build-arg KARAF_VERSION=4.2.0 -t "karaf:4.2.0" karaf
```

Run

- Run Karaf in interactive mode

```
docker run -i -t --name karaf karaf:latest karaf
```

- Run Karaf without interaction but log displayed

```
docker run --name karaf karaf:latest
```

- Kill Karaf

```
docker kill karaf
```

Ports

- The Karaf SSH server is on 8101
- The Karaf WebContainer is on 8888
- The Karaf JMX MBean server is on 1099 (default, not exposed to host) and 44444 (default, not exposed to host)

Edit the `docker-compose.yml` file to edit port settings.

4.15.2. Docker feature

Docker is an optional feature from the Karaf Enterprise features repository.

It means that you have to install the `docker` feature first:

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

The Karaf Docker feature uses the Docker HTTP layer to communicate with the Docker backend. It could be on the same local machine where Apache Karaf instance is running or a remote Docker machine.

The location of the Docker backend (URL) can be specified as an option to the `docker:*` commands. By default, Karaf Docker feature uses `http://localhost:2375`. Please, take a look on the Docker documentation how to enable remote API using HTTP for Docker daemon. In a nutshell, you just have to enable the `tcp` transport connector for the docker daemon. You have to do it using the `-H` option on `dockerd`:

```
/usr/bin/dockerd -H fd:// -H tcp://localhost:2375
```

Apache Karaf Docker feature exposes `DockerService` OSGi service that you can use programmatically (the `docker:*` commands and the `DockerMBean` use the `DockerService` service).

4.15.3. System-wide information

The `docker:info` command provides some details about the docker backend:

```
karaf@root()> docker:info
Containers: 0
Debug: false
Driver: overlay2
ExecutionDriver: null
IPv4Forwarding: true
Images: 1
IndexServerAddress: https://index.docker.io/v1/
InitPath: null
InitSha1: null
KernelVersion: 4.15.0-29-generic
MemoryLimit: true
NEventsListener: false
NFd: 20
NGoroutines: 34
SwapLimit: false
```

4.15.4. Show the Docker version information

The `docker:version` command provides details about the docker version:

```
karaf@root()> docker:version
Version: 17.12.1-ce
Os: linux
Kernel version: 4.15.0-29-generic
Go version: go1.10.1
Git commit: 7390fc6
Arch: amd64
API version: 1.35
Build time: 2018-02-28T17:46:05.000000000+00:00
Experimental: null
```

4.15.5. Search image

The `docker:search` command (or `search()` operation on the `DockerMBean`) searches for a image on Docker HUB:

```
karaf@root()> docker:search java
Name                                        |
Description
           | Automated | Official | Star Count
--------------------------------------------|------------------------------------------------------------
-----------------------------------------------------------|-----------|----------|----------
-
java                                        | Java is a concurrent, class-based, and
object-oriented programming language.                          | false     | true     |
1774
anapsix/alpine-java                         | Oracle Java 8 (and 7) with GLIBC 2.23 over
AlpineLinux                                                    | true      | false    | 332
node                                        | Node.js is a JavaScript-based platform for
server-side and networking applications.                       | false     | true     | 5893
tomcat                                      | Apache Tomcat is an open source
implementation of the Java Servlet and JavaServer Pages technologies | false     |
true     | 1950
openjdk                                     | OpenJDK is an open-source implementation of
the Java Platform, Standard Edition                            | false     | true     | 1097
frekele/java                                | docker run --rm --name java frekele/
java                                                           | true      | false    |
10
ghost                                       | Ghost is a free and open source blogging
platform written in JavaScript                                 | false     | true     | 799
appuio/s2i-maven-java                       | S2I Builder with Maven and
Java                                                                       | true      |
false    | 1
zabbix/zabbix-java-gateway                  | Zabbix Java
Gateway
| true      | false    | 13
jetty                                       | Jetty provides a Web server and
javax.servlet container.                                                   | false     |
true     | 260
fabric8/s2i-java                            | S2I Builder Image for plain Java
applications                                                               | false     |
false    | 5
appuio/s2i-gradle-java                      | S2I Builder with Gradle and
Java                                                                       | true      |
false    | 1
cloudbees/jnlp-slave-with-java-build-tools | Extends cloudbees/java-build-tools docker
image to make it a JNLP slave                                  | true      | false    | 18
blacklabelops/java                          | Java Base
Images.
 | true      | false    | 8
groovy                                      | Apache Groovy is a multi-faceted language
for the Java platform.                                         | false     | true     | 47
lwieske/java-8                              | Oracle Java 8 Container - Full + Slim -
Based off Alpine + CentOS (8u00 - 8u172)                       | true      | false    | 39
davidcaste/alpine-java-unlimited-jce        | Oracle Java 8 (and 7) with GLIBC 2.21 over
```

```
AlpineLinux with unlimited JCE patch applied              | true      | false    | 11
cfje/java-test-applications               | Java Test Applications CI
Image                                                                  | false     |
false    | 0
thingswise/java-docker                    | Java +
dcd
    | true      | false    | 0
rightctrl/java                            | Oracle
Java
    | true      | false    | 2
cfje/java-resource                        | Java Concourse
Resource                                                                |
false     | false    | 0
cfje/java-buildpack                       | Java Buildpack CI
Image                                                                  |
false     | false    | 0
tomee                                     | Apache TomEE is an all-Apache Java EE
certified stack where Apache Tomcat is top dog.                | false     | true     | 53
couchdb                                   | CouchDB is a database that uses JSON for
documents, an HTTP API, & JavaScript/declarative indexing.  | false     | true     | 218
dwolla/java                               | Dwolla's custom Java
image                                                                  |
true      | false    | 1
```

4.15.6. Pull image

The `docker:pull` command (or `pull()` operation on the `DockerMBean`) pull a image from Docker HUB:

```
karaf@root()> docker:pull -v java:8-jre-alpine
{"status":"Pulling from library/java","id":"latest"}
{"status":"Pulling fs layer","progressDetail":{},"id":"5040bd298390"}
{"status":"Pulling fs layer","progressDetail":{},"id":"fce5728aad85"}
{"status":"Pulling fs layer","progressDetail":{},"id":"76610ec20bf5"}
{"status":"Pulling fs layer","progressDetail":{},"id":"60170fec2151"}
{"status":"Pulling fs layer","progressDetail":{},"id":"e98f73de8f0d"}
{"status":"Pulling fs layer","progressDetail":{},"id":"11f7af24ed9c"}
{"status":"Pulling fs layer","progressDetail":{},"id":"49e2d6393f32"}
{"status":"Pulling fs layer","progressDetail":{},"id":"bb9cdec9c7f3"}
{"status":"Waiting","progressDetail":{},"id":"11f7af24ed9c"}
{"status":"Waiting","progressDetail":{},"id":"49e2d6393f32"}
{"status":"Waiting","progressDetail":{},"id":"bb9cdec9c7f3"}
{"status":"Waiting","progressDetail":{},"id":"60170fec2151"}
{"status":"Waiting","progressDetail":{},"id":"e98f73de8f0d"}
```

4.15.7. Listing images

The `docker:images` command (or `images()` operation on the `DockerMBean`) lists the available images on docker:

```
karaf@root()> docker:images
Id                                                                      |
RepoTags              | Created     | Labels | Size       | Virtual Size
────────────────────────────────────────────────────────────────────────┼─────────────────
────┼─────────────┼────────┼───────────┼─────────────
sha256:fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7 | [java:8-jre-
alpine] | 1488578492 | {}     | 107854045 | 107854045
sha256:d23bdf5b1b1b1afce5f1d0fd33e7ed8afbc084b594b9ccf742a5b27080d8a4a8 |
[java:latest]       | 1484614374 | {}     | 643195347 | 643195347
```

4.15.8. Remove image

The `docker:rmi` command (or `rmi()` operation on the `DockerMBean`) removes an image from docker:

```
karaf@root()> docker:rmi --force
sha256:d23bdf5b1b1b1afce5f1d0fd33e7ed8afbc084b594b9ccf742a5b27080d8a4a8
```

4.15.9. Image history

The `docker:history` command displays the complete history for a given image:

```
karaf@root()> docker:history
sha256:fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7
ID                                                                  | Created    |
Created
By

                                | Tags | Size
____________________________________________________________________|____________|____
____________________________________________________________________________________
____________________________________________________________________________________
_____________________________________|______|____________________
sha256:fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7 | 1488578492 |
/bin/sh -c set -x        && apk add --no-cache
openjdk8-jre="$JAVA_ALPINE_VERSION"     && [ "$JAVA_HOME" = "$(docker-java-home)"
]
       |      | [java:8-jre-alpine]
<missing>                                                           | 1488578488 |
/bin/sh -c #(nop)  ENV
JAVA_ALPINE_VERSION=8.111.14-r0

               |      |
<missing>                                                           | 1488578488 |
/bin/sh -c #(nop)  ENV
JAVA_VERSION=8u111

               |      |
<missing>                                                           | 1488578487 |
/bin/sh -c #(nop)  ENV PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/lib/jvm/java-1.8-openjdk/jre/bin:/usr/lib/jvm/java-1.8-openjdk/
bin                                                                 |      |
<missing>                                                           | 1488578487 |
/bin/sh -c #(nop)  ENV JAVA_HOME=/usr/lib/jvm/java-1.8-openjdk/
jre
                                                                    |      |
<missing>                                                           | 1488578458 |
/bin/sh -c {             echo '#!/bin/sh';               echo 'set -e';
echo;           echo 'dirname "$(dirname "$(readlink -f "$(which javac || which
java)")")"';    } > /usr/local/bin/docker-java-home       && chmod +x /usr/local/bin/
docker-java-home |      |
<missing>                                                           | 1488578457 |
/bin/sh -c #(nop)  ENV
LANG=C.UTF-8

               |      |
<missing>                                                           | 1488573141 |
/bin/sh -c #(nop) ADD
file:3df55c321c1c8d73f22bc69240c0764290d6cb293da46ba8f94ed25473fb5853 in
/
                                |      |
```

4.15.10. Pushing and tagging image

The `docker:push` command allows you to push an image on a given repository.

The `docker:tag` command create a new tag for a given image.

4.15.11. Create container

The Karaf Docker feature can create a Docker container based on a given image.

You can use either the `docker:create` shell command or the `create()` operation on the JMX `DockerMBean`.

For instance, here's an example of the `docker:bootstrap` to create a Docker container based on `elasticsearch` instance:

```
karaf@root()> docker:create --image
fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7 --cmd /bin/bash test
```

4.15.12. Listing containers

You can list the containers:

```
karaf@root()> docker:ps -a
Id                                                                 | Names    | Command    |
Created     | Image                                                          | Image
ID                                                                  | Status   | State    |
Ports | Size | Size Root
─────────────────────────────────────────────────────────────────────┼──────────┼────────────┼──
──────────┼──────────────────────────────────────────────────────────────────┼──────────
──────────────────────────────────────────────────────────┼──────────┼──────────┼──────┼
──────┼──────────
92f8b280a7fa69c9ff673ed9678b7040a56c16c9c4aa403498a538cf0f501e9e | [/test] | /bin/bash |
1532809485 | fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7 |
sha256:fdc893b19a147681ee764b2edab6c494d60fe99d83b14b8794bbcbc040ec7aa7 | Created |
created |        | 0    | 0
```

You can also use the containers attribute on the `DockerMBean` JMX MBean or the `containers()` method on the `DockerService` service.

4.15.13. Provision Docker container

Provisioning is a specific way of creating a container based on the current running Karaf instance: it creates a Docker container using the current running Apache Karaf instance `karaf.base`.

You can then reuse this container to create a Docker image and to duplicate the container on another Docker backend via dockerhub.

```
karaf@root()> docker:provision my-karaf
```

You can also use the `provision()` method on the `DockerMBean` JMX MBean or the `DockerService` service.

4.15.14. Start container

You can start a container using the `docker:start` command:

```
karaf@root()> docker:start my-container
```

You can also use the `start()` method on the `DockerMBean` JMX MBean or the `DockerService` service.

4.15.15. Stop container

You can stop a container using the `docker:stop` command:

```
karaf@root()> docker:stop my-container
```

You can also use the `stop()` method on the `DockerMBean` JMX MBean or the `DockerService` service.

4.15.16. Restart container

You can restart a container using the `docker:restart` command:

```
karaf@root()> docker:restart my-container
```

4.15.17. Delete container

You can delete an existing Docker container using the `docker:rm` commmand:

```
karaf@root()> docker:rm my-container
```

You can also use the `rm()` method on the `DockerMBean` JMX MBean or the `DockerService` service.

4.15.18. Pause container

The `docker:pause` command pauses all processes within one or more containers:

```
karaf@root()> docker:pause my-container
```

4.15.19. Unpause container

The `docker:unpause` command unpauses all processes within one or more containers:

```
karaf@root()> docker:unpause my-container
```

4.15.20. Kill container

The `docker:kill` command kills a running container:

```
karaf@root()> docker:kill my-container
```

4.15.21. Rename container

The `docker:rename` command renames an existing container:

```
karaf@root()> docker:rename my-container new-container
```

4.15.22. Logs

The `docker:logs` command displays the log on an existing container:

```
karaf@root()> docker:logs --timestamps --details --stdout --stderr my-container
```

4.15.23. Top

The `docker:top` command displays the current running processes in an existing container:

```
karaf@root()> docker:top my-container
```

4.16. OBR

The goal of OBR (OSGi Bundle Repository) is:

1. to simplify deploying and using bundles
2. to encourage independent bundle development.

OBR achieves the first goal by providing a service that can automatically install a bundle, with its deployment dependencies, from a bundle repository. This makes it easier for people to experiment with existing bundles.

The second goal is achieved by raising the visibility of the available bundles in a repository.

OBR is an optional Apache Karaf feature. You have to install the `obr` feature to use OBR service:

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

The OBR feature turns Apache Karaf into an OBR client. It means that Apache Karaf can use a OBR repository for the installation of the bundles, and during the installation of the features.

The installation of the `obr` feature adds in Apache Karaf:

- the OBR service
- the features OBR resolver
- the `obr:*` commands
- the JMX ObrMBean

The OBR repository contains all bundles. The OBR service knows all requirements and capabilities of each bundle on an OBR repository (it's the OBR metadata).

Thanks to that, when you install ("deploy" in OBR wording) a bundle using the OBR service, it looks for all bundles providing the capabilities matching the bundle requirements. It will automatically install the bundles needed for the bundle.

4.16.1. Features OBR resolver

If the feature specifies `obr` in the `resolver` attribute, Apache Karaf can use the OBR service to construct the list of bundles to install with the features.

The feature default resolver just consider the bundles described in the feature itself.

Using the OBR resolver, Apache Karaf can extend the bundle list at the feature installation time using the OBR service.

4.16.2. Commands

`obr:url-add`

The `obr:url-add` command registers the OBR repository at a given URL in the OBR service.

Basically, an OBR repository is described by a `repository.xml` file.

The `obr:url-add` command expects an `url` argument. The `url` argument is the URL to the OBR repository `repository.xml` file. Any URL described in the [Artifacts repositories and URLs section|urls] is supported.

For instance:

```
karaf@root()> obr:url-add file:///user/.m2/repository/repository.xml
```

`obr:url-list`

The `obr:url-list` command lists the OBR repository (with URL) registered in the OBR service:

```
karaf@root()> obr:url-list
Index | OBR URL
-----------------------------------------------------------
0     | file:/user/.m2/repository/repository.xml
```

`obr:url-refresh`

The `obr:url-refresh` command refreshes an OBR repository (reloading the URL).

The OBR service doesn't take "on the fly" the changes performed on an OBR repository `repository.xml`. You have to reload the `repository.xml` URL to take the changes. It's the purpose of the `obr:url-refresh` command.

Without argument, the `obr:url-refresh` command refreshes all repositories:

```
karaf@root()> obr:url-refresh
```

You can refresh only one repository by specifying the URL as argument:

```
karaf@root()> obr:url-refresh file:/user/.m2/repository/repository.xml
```

Instead of using the URL, you can use the repository index as displayed by the `obr:url-list` command. To do so, you have to use the `-i` option:

```
karaf@root()> obr:url-refresh -i 0
```

`obr:url-remove`

The `obr:url-remove` command removes an OBR repository from the OBR service.

The `obr:url-remove` command expects the repository URL as argument:

```
karaf@root()> obr:url-remove file:/user/.m2/repository/repository.xml
```

Instead of using the URL, you can use the repository index as displayed by the `obr:url-list` command. To do so, you have to use the `-i` option:

```
karaf@root()> obr:url-remove -i 0
```

`obr:list`

The `obr:list` command lists all bundles available on the registered OBR repositories:

```
karaf@root()> obr:list|more
Name                                                                                | Symbolic
Name                                                                          | Version
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
activemq-blueprint                                                                  |
org.apache.activemq.activemq-blueprint                                        |
5.12.0.SNAPSHOT
activemq-camel                                                                      |
org.apache.activemq.activemq-camel                                            |
5.12.0.SNAPSHOT
activemq-karaf                                                                      | activemq-
karaf                                                                         | 5.12.0.SNAPSHOT
activemq-osgi                                                                       |
org.apache.activemq.activemq-osgi                                             |
5.12.0.SNAPSHOT
Apache Aries Application API                                                        |
org.apache.aries.application.api                                              | 1.0.1.SNAPSHOT
...
```

`obr:info`

The `obr:info` command displays the details about bundles available on the OBR service. Especially, it provides details about capabilities and requirements of bundles.

The `obr:info` command expects a bundle symbolic name as argument:

```
karaf@root()> obr:info org.apache.karaf.wrapper.core
------------------------------
Apache Karaf :: Wrapper :: Core
------------------------------
id: org.apache.karaf.wrapper.core/4.0.0
description: Core implementation and integration of the Java Service Wrapper.        It
provides a complete integration of Karaf with your Operating System.
documentation: http://www.apache.org/
symbolicname: org.apache.karaf.wrapper.core
presentationname: Apache Karaf :: Wrapper :: Core
license: http://www.apache.org/licenses/LICENSE-2.0.txt
uri: file:/user/.m2/repository/org/apache/karaf/wrapper/org.apache.karaf.wrapper.core/
4.0.0/org.apache.karaf.wrapper.core-4.0.0.jar
size: 1281352
version: 4.0.0
Requires:

service:(&(service=org.apache.aries.blueprint.NamespaceHandler)(osgi.service.blueprint.nam
espace=http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0))
   package:(&(package=javax.management))
   package:(&(package=org.apache.karaf.wrapper))
   package:(&(package=org.apache.karaf.wrapper.management))
   package:(&(package=org.fusesource.jansi)(version>=1.11.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework)(version>=1.7.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework.launch)(version>=1.1.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework.startlevel)(version>=1.0.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.service.blueprint)(version>=1.0.0)(!(version>=2.0.0)))
   package:(&(package=org.slf4j)(version>=1.7.0)(!(version>=2.0.0)))
Capabilities:
   bundle:{manifestversion=2, symbolicname=org.apache.karaf.wrapper.core,
presentationname=Apache Karaf :: Wrapper :: Core, version=4.0.0}
   service:{service=org.apache.karaf.wrapper.WrapperService}
   package:{package=org.apache.karaf.wrapper, version=4.0.0}
   package:{package=org.apache.karaf.wrapper.management, uses:=javax.management,
version=4.0.0}
```

The `obr:info` command uses the following syntax to identify the bundles: `symbolic_name,version` where `version` is optional. It means that you have to use the following command to see the info about the wrapper core bundle with version 4.0.0:

```
karaf@root()> obr:info org.apache.karaf.wrapper.core,4.0.0
...
```

You can specific multiple bundles separated by space:

```
karaf@root()> obr:info org.apache.karaf.wrapper.core,4.0.0
org.apache.karaf.wrapper.command,4.0.0
...
```

`obr:source`

In addition of the bundles executable, the OBR service can also store the bundles sources.

The `obr:source` command check the source URL in the OBR metadata for a given bundle, and download the sources on a target folder:

```
karaf@root()> obr:source /tmp org.apache.karaf.wrapper.core
```

The `obr:source` command uses the following syntax to identify the bundles: `symbolic_name,version` where `version` is optional. It means that you have to use the following command to download the source of wrapper core bundle with version 4.0.0:

```
karaf@root()> obr:source /tmp org.apache.karaf.wrapper.core,4.0.0
```

You can specify multiple bundles separated by space:

```
karaf@root()> obr:source /tmp org.apache.karaf.wrapper.core,4.0.0
org.apache.karaf.wrapper.command,4.0.0
...
```

`obr:resolve`

The `obr:resolve` command displays the resolution output for a given set of requirements. Actually, it show the bundles providing the capabilities to match the requirements. It's what the OBR service does when executing `obr:deploy`.

Optionally, the `obr:resolve` command can deploy the bundles as the `obr:deploy` command does.

For instance, to know the OBR bundle resolving the `org.apache.karaf.wrapper` package requirement, you can do:

```
karaf@root()> obr:resolve package=org.apache.karaf.wrapper
Required resource(s):
---------------------
   Apache Karaf :: Wrapper :: Core (4.0.0)
```

`obr:find`

The `obr:find` command is similar to the `obr:resolve` one. It displays the bundles resolving the provided requirements, with details.

For instance, to find the OBR bundle providing the `org.apache.karaf.wrapper` package, you can do:

```
karaf@root()> obr:find package=org.apache.karaf.wrapper
------------------------------
Apache Karaf :: Wrapper :: Core
------------------------------
id: org.apache.karaf.wrapper.core/4.0.0
description: Core implementation and integration of the Java Service Wrapper.        It
provides a complete integration of Karaf with your Operating System.
documentation: http://www.apache.org/
symbolicname: org.apache.karaf.wrapper.core
presentationname: Apache Karaf :: Wrapper :: Core
license: http://www.apache.org/licenses/LICENSE-2.0.txt
uri: file:/user/.m2/repository/org/apache/karaf/wrapper/org.apache.karaf.wrapper.core/
4.0.0/org.apache.karaf.wrapper.core-4.0.0.jar
size: 1281352
version: 4.0.0
Requirements:

service:(&(service=org.apache.aries.blueprint.NamespaceHandler)(osgi.service.blueprint.nam
espace=http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0))
   package:(&(package=javax.management))
   package:(&(package=org.apache.karaf.wrapper))
   package:(&(package=org.apache.karaf.wrapper.management))
   package:(&(package=org.fusesource.jansi)(version>=1.11.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework)(version>=1.7.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework.launch)(version>=1.1.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.framework.startlevel)(version>=1.0.0)(!(version>=2.0.0)))
   package:(&(package=org.osgi.service.blueprint)(version>=1.0.0)(!(version>=2.0.0)))
   package:(&(package=org.slf4j)(version>=1.7.0)(!(version>=2.0.0)))
Capabilities:
   bundle:{manifestversion=2, symbolicname=org.apache.karaf.wrapper.core,
presentationname=Apache Karaf :: Wrapper :: Core, version=4.0.0}
   service:{service=org.apache.karaf.wrapper.WrapperService}
   package:{package=org.apache.karaf.wrapper, version=4.0.0}
   package:{package=org.apache.karaf.wrapper.management, uses:=javax.management,
version=4.0.0}
```

`obr:deploy`

The `obr:deploy` command installs a bundle from the OBR repository, including all bundles required to satisfy the bundle requirements.

```
karaf@root()> obr:deploy org.ops4j.pax.web.samples.helloworld-hs
Target resource(s):
-------------------
   OPS4J Pax Web - Samples - Hello World - HttpService (4.0.0.SNAPSHOT)

Required resource(s):
---------------------
   Apache ServiceMix :: Specs :: Activation API 1.4 (2.3.0.SNAPSHOT)
   OPS4J Pax Web - Jetty Bundle (4.0.0.SNAPSHOT)

Deploying...done.
```

By default, the bundles are just installed, not started. You can use the `-s` option to start the bundles.

The `obr:deploy` command uses the following syntax to identify the bundles: `symbolic_name,version` where `version` is optional. It means that you have to use the following command to deploy the wrapper core bundle with version 4.0.0:

```
karaf@root()> obr:deploy org.apache.karaf.wrapper.core,4.0.0
```

You can specify multiple bundles separated by space:

```
karaf@root()> obr:deploy org.apache.karaf.wrapper.core,4.0.0
org.apache.karaf.wrapper.command,4.0.0
...
```

4.16.3. `obr:start`

The `obr:start` command does the same as `obr:deploy -s` command. It installs the bundle (and all required bundles to satisfy the requirements) and starts all installed bundles.

```
karaf@root()> obr:start org.ops4j.pax.web.samples.helloworld-hs
Target resource(s):
-------------------
   OPS4J Pax Web - Samples - Hello World - HttpService (4.0.0.SNAPSHOT)

Required resource(s):
---------------------
   Apache ServiceMix :: Specs :: Activation API 1.4 (2.3.0.SNAPSHOT)
   OPS4J Pax Web - Jetty Bundle (4.0.0.SNAPSHOT)

Deploying...done.
```

The `obr:start` command uses the following syntax to identify the bundles: `symbolic_name,version` where `version` is optional. It means that you have to use the following command to deploy and start the wrapper core bundle with version 4.0.0:

```
karaf@root()> obr:start org.apache.karaf.wrapper.core,4.0.0
```

You can specify multiple bundles separated by space:

```
karaf@root()> obr:start org.apache.karaf.wrapper.core,4.0.0
org.apache.karaf.wrapper.command,4.0.0
...
```

4.16.4. JMX ObrMBean

On the JMX layer, Apache Karaf provides a MBean dedicated to the management of the OBR service: the ObrMBean.

The ObjectName to use is `org.apache.karaf:type=obr,name=*`.

Attributes

The `Urls` attribute provides the list of registered OBR repositories URLs.

The `Bundles` attribute provides a tabular data containing all bundles available on the registered OBR repositories.

Operations

- `addUrl(url)` registers the OBR repository using the `url` to the `repository.xml`.
- `removeUrl(url)` removes the OBR repository at the given `url`.

- `refreshUrl(url)` refreshes the OBR repository at the given `url`.
- `deployBundle(name)` deploys a bundle (and all bundles required to satisfy the requirements) using the OBR service. The bundles are not automatically started.
- `deployBundle(name, start, deployOptional)` deploys a bundle (and all bundles required to satisfy the requirements) using the OBR service. If `start` is `true`, the bundles are automatically started. If `deployOptional` is `true`, even the optional requirements will be resolved by the OBR service (meaning installing more bundles to satisfy the optional requirements).

The name to identify a bundle uses the following syntax: `symbolic_name,version` where version is optional.

4.16.5. Apache Karaf Cave

In addition of being an OBR client, Apache Karaf can act as an OBR repositories server, thanks to Apache Karaf Cave.

See the https://karaf.apache.org/projects.html for details.

4.17. Enterprise

4.17.1. Http Service

Apache Karaf supports several Http Services:

- `http` feature installs Http Service powered by Pax Web
- `felix-http` feature installs Http Service powered by Apache Felix HTTP

Installing the Http Service

If you want to use Pax Web Http Service, you have to install the `http` feature:

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

NB: the HTTP service will be actually bound only when at least one web application will use it.

If you want to use Apache Felix Http Service, you have to install `felix-http` feature:

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

If you want to test the HTTP service, you can install the `webconsole` feature:

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

Then, you can test the HTTP service by accessing the Apache Karaf WebConsole pointing your browser to [http://localhost:8181/system/console].

Configuring the HTTPService

By default the HTTPService listens on port 8181 you can change the port by creating a file `etc/org.ops4j.pax.web.cfg` (if you use the Pax Web Http Service) with the following content:

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

or by typing:

```
root@karaf> config:property-set -p org.ops4j.pax.web org.osgi.service.http.port 8181
```

If the http feature is already installed the change will take effect immediately.

If you use Apache Felix Http Service, it's basically the same but the configuration is `etc/org.apache.felix.http.cfg`.

Registering a servlet with the HttpService manually

See the Apache Karaf Servlet example: https://github.com/apache/karaf/tree/master/examples/karaf-servlet-example/karaf-servlet-example-registration

Using the Pax Web whiteboard extender

The Pax Web whiteboard extender is an enhancement of the http feature. So use the following command to install:

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

NB: `felix-http` feature installs whiteboard by default in the feature.

You can take a look on a full example using whiteboard here: https://github.com/apache/karaf/tree/master/examples/karaf-servlet-example

For commands take a look at the command section in the webcontainer chapter.

4.17.2. WebContainer (JSP/Servlet)

Apache Karaf can act as a complete WebContainer, fully supporting the JSP/Servlet specifications.

Apache Karaf WebContainer supports both:

- WAB (WebApplication Bundles) which are OSGi native web applications
- WAR (WebApplication aRchives) which are non-OSGi web applications (the same as you can deploy in any web container like Apache Tomcat)

To enable the Apache Karaf WebContainer, you just have to install the `war` feature:

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

> **NOTE** The installation of the `webconsole` feature automatically installs the `war` feature.

The `war` feature provides:

- an embedded web container (powered by Jetty), with its configuration
- a set of console commands
- a new war deployer

Configuration

The default port used by the WebContainer is 8181. Note: the connector is actually bound only when at least a servlet or webapplication is using it. It means that just installing the `http` or `war` feature doesn't bind the connector.

By default, Karaf creates an internal Jetty connector that you can configure via `etc/org.ops4j.pax.web.cfg`:

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

Note: if you want to use port numbers < 1024, remember you have to run with root privileges. However note that this is not a good idea from a security point of view.

It’s possible to enable the HTTPs "internal" connector. The first step is to create a keystore containing a server certificate. For instance the following command creates a keystore with a self-signed certificate:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore -storepass karaf1234
-validity 360 -keysize 2048
```

Now, we can enable and configure the HTTPs connector with this keystore in `etc/org.ops4j.pax.web.cfg`:

```
org.osgi.service.http.port.secure=8443
org.osgi.service.http.secure.enabled=true
org.ops4j.pax.web.ssl.keystore=/path/to/keystore
org.ops4j.pax.web.ssl.password=foo
org.ops4j.pax.web.ssl.keypassword=karaf1234
```

It’s possible to use only HTTPs and to disable the HTTP using:

```
org.osgi.service.http.enabled=false
```

```
org.osgi.service.https.enabled=true
```

As an alternative to the default connectors, it is possible to configure additional connectors in the `etc/jetty.xml` configuration file.

The `etc/jetty.xml` is a standard Eclipse Jetty configuration file.

The default Apache Karaf WebContainer `etc/jetty.xml` contains:

```
<?xml version="1.0"?>
<!--
 Licensed to the Apache Software Foundation (ASF) under one
 or more contributor license agreements.  See the NOTICE file
 distributed with this work for additional information
 regarding copyright ownership.  The ASF licenses this file
 to you under the Apache License, Version 2.0 (the
 "License"); you may not use this file except in compliance
 with the License.  You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing,
 software distributed under the License is distributed on an
 "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 KIND, either express or implied.  See the License for the
 specific language governing permissions and limitations
 under the License.
-->
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//
DTD Configure//EN" "http://jetty.mortbay.org/configure.dtd">

<Configure class="org.eclipse.jetty.server.Server">

    <!-- =========================================================== -->
    <!-- Set connectors -->
    <!-- =========================================================== -->
    <!-- One of each type! -->
    <!-- =========================================================== -->

    <!-- Use this connector for many frequently idle connections and for
        threadless continuations. -->
    <Call name="addConnector">
        <Arg>
            <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
                <Set name="host">
                    <Property name="jetty.host" />
                </Set>
                <Set name="port">
                    <Property name="jetty.port" default="8181" />
                </Set>
                <Set name="maxIdleTime">300000</Set>
                <Set name="Acceptors">2</Set>
                <Set name="statsOn">false</Set>
                <Set name="confidentialPort">8443</Set>
                <Set name="lowResourcesConnections">20000</Set>
                <Set name="lowResourcesMaxIdleTime">5000</Set>
            </New>
        </Arg>
```

```
    </Call>

    <!-- ============================================================= -->
    <!-- Configure Authentication Realms -->
    <!-- Realms may be configured for the entire server here, or -->
    <!-- they can be configured for a specific web app in a context -->
    <!-- configuration (see $(jetty.home)/contexts/test.xml for an -->
    <!-- example). -->
    <!-- ============================================================= -->
    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.jetty.plus.jaas.JAASLoginService">
                <Set name="name">karaf</Set>
                <Set name="loginModuleName">karaf</Set>
                <Set name="roleClassNames">
                    <Array type="java.lang.String">
                        <Item>org.apache.karaf.jaas.boot.principal.RolePrincipal
                        </Item>
                    </Array>
                </Set>
            </New>
        </Arg>
    </Call>
    <Call name="addBean">
        <Arg>
            <New class="org.eclipse.jetty.plus.jaas.JAASLoginService">
                <Set name="name">default</Set>
                <Set name="loginModuleName">karaf</Set>
                <Set name="roleClassNames">
                    <Array type="java.lang.String">
                        <Item>org.apache.karaf.jaas.boot.principal.RolePrincipal
                        </Item>
                    </Array>
                </Set>
            </New>
        </Arg>
    </Call>

</Configure>
```

The `SelectChannelConnector` defines the default connector of the WebContainer.

This connector defines the 8181 port number for the HTTP protocol (`port` property), and the 8443 port number for the HTTPS protocol (`confidentialPort` property).

By default, Apache Karaf bind these ports on all network interfaces (`0.0.0.0`). You can config the `host` property to bind on a specific network interface (with a given IP address).

The following resources give you details about advanced `etc/jetty.xml` configurations:

- http://wiki.eclipse.org/Jetty/Howto/Configure_Jetty
- http://wiki.eclipse.org/Jetty/Howto/Configure_SSL
- http://wiki.eclipse.org/Jetty/Reference/jetty.xml_syntax

Deploy

Apache Karaf WebContainer is able to deploy:

- pure OSGi WebApplication Bundle (WAB)
- "classical" standard WebApplication aRchive (WAR)

WAB (WebApplication Bundle)

A WAB is a standard WAR or JAR archive containing at least the following properties in the MANIFEST:

- `Bundle-ManifestVersion: 2` defines that the bundle follows the rules of R4 specification.
- `Bundle-SymbolicName` specifies a unique, non-localizable name for the bundle. This name should be based on the reverse domain name convention.
- `Web-ContextPath` specifies the location of the web application.

WAB can be deployed directly in Apache Karaf, for instance, by dropping the archive in the `deploy` folder, or using the `bundle:install` command.

For instance, the Apache Karaf manual (documentation) is available as a WAB that you can deploy directly in a running instance:

```
karaf@root()> bundle:install -s mvn:org.apache.karaf/manual/4.0.0/war
```

WAR (WebApplication aRchive)

Apache Karaf allows you to deploy directly WAR files without repackaging as WAB.

Using the `webbundle` prefix and providing headers directly on the URL, Apache Karaf creates a WAB "on the fly".

For instance, you can deploy the Apache Tomcat sample non-OSGi "classical" WAR with the following command:

```
karaf@root()> bundle:install -s "webbundle:http://tomcat.apache.org/tomcat-7.0-doc/appdev/
sample/sample.war?Bundle-SymbolicName=tomcat-sample&Web-ContextPath=/sample"
```

You can note the `webbundle` prefix, and the `Bundle-SymbolicName` and `Web-ContextPath` headers on the URL.

HTTP proxy

Apache Karaf provides a HTTP proxy service. It allows you to proxy any HTTP URLs within Karaf. It allows you to expose remote web applications in Karaf.

You can use the Karaf `ProxyService` programmatically, or via the corresponding shell commands and MBeans.

Commands

`http:list`

The `http:list` lists the available Servlets deployed in the WebContainer.

For instance, if you have installed the Apache Karaf WebConsole, you can see the WebConsole Servlets:

```
karaf@root()> http:list
ID  | Servlet          | Servlet-Name   | State       | Alias               | Url
-------------------------------------------------------------------------------------
-----------
113 | ResourceServlet  | /res           | Deployed    | /system/console/res | [/system/
console/res/*]
113 | KarafOsgiManager | ServletModel-2 | Undeployed  | /system/console     | [/system/
console/*]
113 | KarafOsgiManager | ServletModel-5 | Deployed    | /system/console     | [/system/
console/*]
```

The `ID` is the ID of the bundle which provides the servlet (`113` here).

The `State` is the current state of the Servlet (`Deployed` or `Undeployed`).

The `Url` is the URL where the Servlet is available.

`web:list`

The `web:list` command lists the WebApplication Bundles ("native" WAB or "wrapped WAR") deployed in the WebContainer.

For instance, if you installed the Apache Karaf manual WAR file as described previously, you can see it with `web:list`:

```
karaf@root()> web:list
ID  | State       | Web-State   | Level | Web-ContextPath | Name
-------------------------------------------------------------------------------------
---------
111 | Active      | Deployed    | 80    | /karaf-doc      | Apache Karaf :: Manual (4.0.0)
```

`web:stop`

The `web:stop` command stops a web application in the WebContainer. The `web:stop` command expects a `id` argument corresponding to the bundle ID (as displayed by the `web:list` command).

For instance, to stop the Apache Karaf manual web application:

```
karaf@root()> web:stop 111
```

`web:start`

The `web:start` command starts a web application in the WebContainer. The `web:start` command expects a `id` argument corresponding to the bundle ID (as displayed by the `web:list` command).

For instance, to start the Apache Karaf manual web application:

```
karaf@root()> web:start 111
```

`http:proxy-list`

The `http:proxy-list` command list the configured HTTP proxies:

```
karaf@root()> http:proxy-list
URL          | ProxyTo                                  | Balancing Policy
-------------|------------------------------------------|-----------------
/webconsole | http://localhost:8181/system/console |
```

`http:proxy-add` and `http:proxy-balancing-list`

The `http:proxy-add` registers a new HTTP proxy. For instance, you can proxy the Karaf WebConsole on another URL of your choice using:

```
karaf@root()> http:proxy-add /webconsole http://localhost:8181/system/console
```

Karaf HTTP Proxy can proxy any URL, like a backend running on Docker or a remote URL.

It's also possible to proxy several URLs, defining a balancing policy. By default, two balancing policies are available: random (selecting one URL randomly) and round-robin (selecting one URL after another one). It's possible to create your own balancing policy by implementing a `BalancingPolicy` service (with the `type` service property).

You can see the balancing policies available using `http:proxy-balancing-list` command:

```
karaf@root()> http:proxy-balancing-list
random
round-robin
```

Then, you can use add a proxy with several targets and a policy:

```
karaf@root()> http:proxy-add -b round-robin /my http://host1/my,http://host2/
my,http://host3/my
```

You can see the list and balancing policy in used using `http:proxy-list`:

```
karaf@root()> http:proxy-list
URL          | ProxyTo                                              | Balancing Policy
-------------|------------------------------------------------------|-----------------
/my          | http://host1/my,http://host2/my,http://host3/my | round-robin
```

`http:proxy-remove`

The `http:proxy-remove` removes an existing HTTP proxy:

```
karaf@root()> http:proxy-remove /webconsole
```

JMX HttpMBean

On the JMX layer, you have a MBean dedicated to the manipulation of the Servlets: the HttpMBean.

The ObjectName to use is `org.apache.karaf:type=http,name=*`.

Attributes

The `Servlets` attribute provides a tabular data providing the list of deployed Servlets including:

- `Alias` is the Servlet URL alias.
- `Bundle-ID` is the ID of the bundle which provides this Servlet.
- `Servlet` is the class name of the Servlet.
- `State` is the current Servlet state (`Deployed` or `Undeployed`).
- `URL` is the URL of the Servlet (the Servlet context path).

The `Proxies` attribute provides a tabular data providing the list of HTTP proxies including:

- `URL` is the proxy URL.
- `proxyTo` is the proxy target.
- `prefix` is optional proxy prefix.

The `ProxyBalacingPolicies` attribute provides the collection of balancing policies available.

Operations

- `addProxy(url, proxyTo, prefix)` registers a new HTTP proxy.
- `removeProxy(url)` removes an existing HTTP proxy.

JMX WebMBean

On the JMX layer, you have a MBean dedicated to the manipulation of the Web Applications: the WebMBean.

The ObjectName to use is `org.apache.karaf:type=web,name=*`.

Attributes

The `WebBundles` attribute provides a tabular data providing the list of deployed Web Applications including:

- `ID` is the ID of the bundle providing the Web Application.
- `Level` is the bundle start level.
- `Name` is the bundle symbolic name providing the Web Application.
- `State` is the current state of the bundle.
- `Web-ContextPath` is the context path of the Web Application.
- `Web-State` is the current status of the Web Application (`Deployed` or `Undeployed`).

Operations

- `start(id)` starts the web context of the bundle with `id`.
- `start(list)` starts the web context of the bundles with ID in the provided `list`.
- `stop(id)` stops the web context of the bundle with `id`.
- `stop(list)` stops the web context of the bundles with ID in the provided `list`.

4.17.3. Naming (JNDI)

The Apache Karaf Naming (JNDI) is an optional enterprise feature.

You have to install the `jndi` feature first:

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

Apache Karaf provides a complete JNDI support.

You have two parts in the Apache Karaf JNDI support:

- a fully compliant implementation of the OSGi Alliance JNDI Service specification.
- a more "regular" JNDI context, containing different names that you can administrate.

OSGi Services Registry and JNDI

The OSGi Service Registry provides a centralized register/query capabilities for OSGi services.

A common pattern outside of OSGi is to make use of JNDI API to access services from a directory system. The OSGi service registry can be viewed as an example of such a system.

Apache Karaf supports the `osgi:service` lookup scheme as defined by the JNDI Service Specification.

The schema is:

```
osgi:service/<interface>[/<filter>]
```

For instance, you can directly use JNDI to get a OSGi service:

```
Context ctx = new InitialContext();
Runnable r = (Runnable) ctx.lookup("osgi:service/java.lang.Runnable");
```

JNDI service

Apache Karaf also supports regular JNDI, including a directoy system where you can register name bindings, sub-context, etc.

It supports the standard JNDI API:

```
Context ctx = new InitialContext();
Runnable r = (Runnable) ctx.lookup("this/is/the/name");
```

It also allows you to bind some OSGi services as "pure" JNDI name. In that case, you don't have to use the specific `osgi:service` scheme.

Commands

Apache Karaf provides specific commands to manipulate the JNDI service.

`jndi:names`

The `jndi:names` command lists all JNDI names. It groups both the JNDI names from the `osgi:service` scheme and the regular JNDI names:

```
karaf@root()> jndi:names
JNDI Name         | Class Name
------------------------------------------------------------------
osgi:service/jndi | org.apache.karaf.jndi.internal.JndiServiceImpl
jndi/service      | org.apache.karaf.jndi.internal.JndiServiceImpl
```

We can see here the `osgi:service/jndi` name (using the `osgi:service` scheme) and `jndi/service` name (using the regular JNDI service).

The `jndi:names` command accepts an optional `context` argument to list names on the given context.

For instance, you can list only names in the `jndi` sub-context:

```
karaf@root()> jndi:names jndi
JNDI Name | Class Name
------------------------------------------------------------
service   | org.apache.karaf.jndi.internal.JndiServiceImpl
```

> **NOTE** The `jndi:names` lists only names (the full qualified name). It means that the empty JNDI sub-contexts are not displayed. To display all JNDI sub-contexts (empty or not), you can use the `jndi:contexts` command.

`jndi:contexts`

The `jndi:contexts` command lists all JNDI sub-contexts:

```
karaf@root()> jndi:contexts
JNDI Sub-Context
----------------
other/context
foo/bar
```

`jndi:create`

The `jndi:create` command creates a new JNDI sub-context:

```
karaf@root()> jndi:create my/company
```

`jndi:delete`

The `jndi:delete` command deletes a JNDI sub-context:

```
karaf@root()> jndi:delete my/company
```

`jndi:alias`

The `jndi:alias` command creates a new JNDI name (alias) with an existing one.

The existing JNDI name can be a regular one:

```
karaf@root()> jndi:alias bean/services/jndi aliases/services/jndi
karaf@root()> jndi:names
JNDI Name             | Class Name
------------------------------------------------------------------------
osgi:service/jndi     | org.apache.karaf.jndi.internal.JndiServiceImpl
bean/services/jndi    | org.apache.karaf.jndi.internal.JndiServiceImpl
aliases/services/jndi | org.apache.karaf.jndi.internal.JndiServiceImpl
```

or a name from the `osgi:service` schema:

```
karaf@root()> jndi:alias osgi:service/jndi alias/jndi/service
karaf@root()> jndi:names
JNDI Name          | Class Name
------------------------------------------------------------------------
osgi:service/jndi  | org.apache.karaf.jndi.internal.JndiServiceImpl
alias/jndi/service | org.apache.karaf.jndi.internal.JndiServiceImpl
```

> **NOTE** The `jndi:alias` automatically creates all required JNDI sub-contexts.

`jndi:bind`

The `jndi:bind` command binds an OSGi service with a JNDI name.

The `jndi:bind` command requires an OSGi service ID and a JNDI name. The OSGi service ID can be found using the `service:list` command.

For instance, we can bind the OSGi service with ID 344 with the JNDI name `services/kar`:

```
karaf@root()> jndi:bind 344 services/kar
karaf@root()> jndi:names
JNDI Name         | Class Name
--------------------------------------------------------------------------------
osgi:service/jndi | org.apache.karaf.jndi.internal.JndiServiceImpl
services/kar      | org.apache.karaf.kar.internal.KarServiceImpl
```

`jndi:unbind`

The `jndi:unbind` command unbind a given JNDI name:

```
karaf@root()> jndi:names
JNDI Name         | Class Name
--------------------------------------------------------------------------------
osgi:service/jndi | org.apache.karaf.jndi.internal.JndiServiceImpl
services/kar      | org.apache.karaf.kar.internal.KarServiceImpl
karaf@root()> jndi:unbind services/kar
karaf@root()> jndi:names
JNDI Name         | Class Name
--------------------------------------------------------------------------------
osgi:service/jndi | org.apache.karaf.jndi.internal.JndiServiceImpl
```

NOTE

It's not possible to unbind a name from the `osgi:service` schema, as it's linked to a OSGi service.

JMX JndiMBean

The JMX JndiMBean provides the JNDI names, and the operations to manipulate the JNDI service.

The object name to use is `org.apache.karaf:type=jndi,name=*`.

Attributes

The `Names` attribute provides a map containing all JNDI names and class names from both `osgi:service` scheme and the regular JNDI service.

The `Contexts` attribute provides a list containing all JNDI sub-contexts.

Operations

- `getNames(context)` provides a map containing JNDI names and class names in a given JNDI sub-context.
- `create(context)` creates a new JNDI sub-context.

- `delete(context)` deletes a JNDI sub-context.
- `alias(name, alias` creates a JNDI name (alias) for a given one.
- `bind(serviceId, name` binds a JNDI name using an OSGi service (identified by its ID).
- `unbind(name)` unbinds a JNDI name.

4.17.4. Transaction (JTA)

Apache Karaf provides container managed transactions, available as OSGi services.

As most of the enterprise features, it's an optional feature that you can install with:

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

However, the `transaction` feature is installed (as a transitive dependency) when installing enterprise features (like `jdbc` or `jms` features for instance).

Apache Aries Transaction and ObjectWeb HOWL

The `transaction` feature uses Apache Aries and ObjectWeb HOWL. Aapache Aries Transaction "exposes" the transaction manager as an OSGi service. The actual implementation of the transaction manager is ObjectWeb HOWL.

ObjectWeb HOWL is a logger implementation providing features required by the ObjectWeb JOTM project, with a public API that is generally usable by any Transaction Manager. ObjectWeb HOWL uses unformatted binary logs to maximize performance and specifies a journalization API with methods necessary to support JOTM recovery operations.

ObjectWeb HOWL is intended to be used for logging of temporary data such as XA transaction events. It is not a replacement for traditional log kits such as LOG4J and Java SE Logging.

In Apache Karaf, ObjectWeb HOWL (High-speed ObjectWeb Logger) is used to implement TransactionLog (in Aries Transaction), providing a very performant transaction manager in an OSGi way.

Configuration

The installation of the `transaction` feature installs a new configuration: `org.apache.aries.transaction`.

You can see the configuration properties using:

```
karaf@root()> config:list "(service.pid=org.apache.aries.transaction)"
----------------------------------------------------------------
Pid:            org.apache.aries.transaction
BundleLocation: mvn:org.apache.aries.transaction/org.apache.aries.transaction.manager/
1.1.0
Properties:
   aries.transaction.recoverable = true
   aries.transaction.timeout = 600
   service.pid = org.apache.aries.transaction
   org.apache.karaf.features.configKey = org.apache.aries.transaction
   aries.transaction.howl.maxBlocksPerFile = 512
   aries.transaction.howl.maxLogFiles = 2
   aries.transaction.howl.logFileDir = /opt/apache-karaf-4.0.0/data/txlog
   aries.transaction.howl.bufferSizeKBytes = 4
```

- `aries.transaction.recoverable` property is a flag to enable support of recoverable resource or not. A recoverable resource is a transactional object whose state is saved to stable storage if the transaction is committed, and whose state can be reset to what it was at the beginning of the transaction if the transaction is rolled back. At commit time, the transaction manager uses the two-phase XA protocol when communicating with the recoverable resource to ensure transactional integrity when more than one recoverable resource is involved in the transaction being committed. Transactional databases and message brokers like Apache ActiveMQ are examples of recoverable resources. A recoverable resource is represented using the javax.transaction.xa.XAResource interface in JTA. Default is `true`.
- `aries.transaction.timeout` property is the transaction timeout. If a transaction has a lifetime longer than this timeout a transaction exception is raised and the transaction is rollbacked. Default is `600` (10 minutes).
- `aries.transaction.howl.logFileDir` property is the directory where the transaction logs (journal) are stored. Default is `KARAF_DATA/txlog`.
- `aries.transaction.howl.maxLogFiles` property is the maximum number of transaction log files to retain. Combined with the `aries.transaction.howl.maxBlocksPerFile`, it defines the transaction retention.

You can change the configuration directly using the `config:*` commands, or the Config MBean.

For instance, to increase the transaction timeout, you can do:

```
karaf@root()> config:edit org.apache.aries.transaction
karaf@root()> config:property-set aries.transaction.timeout 1200
karaf@root()> config:update
karaf@root()> config:list "(service.pid=org.apache.aries.transaction)"
----------------------------------------------------------------
Pid:            org.apache.aries.transaction
BundleLocation: mvn:org.apache.aries.transaction/org.apache.aries.transaction.manager/
1.1.0
Properties:
   aries.transaction.recoverable = true
   aries.transaction.timeout = 1200
   service.pid = org.apache.aries.transaction
   org.apache.karaf.features.configKey = org.apache.aries.transaction
   aries.transaction.howl.maxBlocksPerFile = 512
   aries.transaction.howl.maxLogFiles = 2
   aries.transaction.howl.logFileDir = /opt/apache-karaf-4.0.0/data/txlog
   aries.transaction.howl.bufferSizeKBytes = 4
```

NOTE

The `transaction` feature defines the configuration in memory by default. It means that changes that you can do will be lost in case of Apache Karaf restart. If you want to define your own transaction configuration at startup, you have to create a `etc/org.apache.aries.transaction.cfg` configuration file and set the properties and values in the file. For instance:

```
# etc/org.apache.aries.transaction.cfg
aries.transaction.recoverable = true
aries.transaction.timeout = 1200
aries.transaction.howl.maxBlocksPerFile = 512
aries.transaction.howl.maxLogFiles = 2
aries.transaction.howl.logFileDir = /opt/apache-karaf-4.0.0/data/txlog
aries.transaction.howl.bufferSizeKBytes = 4
```

4.17.5. DataSources (JDBC)

The Apache Karaf DataSources (JDBC) is an optional enterprise feature.

You have to install the following features first:

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

NB: `jdbc` feature automatically installs the `pax-jdbc-*` core features.

Pax JDBC provides ready to use adapters for different databases:

- pax-jdbc-derby
- pax-jdbc-derbyclient
- pax-jdbc-h2
- pax-jdbc-mariadb
- pax-jdbc-mysql
- pax-jdbc-oracle
- pax-jdbc-postgresql
- pax-jdbc-sqlite
- pax-jdbc-mssql

This feature provides an OSGi service to create/delete JDBC datasources in the container and perform database operations (SQL queries).

This JDBC OSGi service can be manipulated programmatically (see the developer guide for details), using the `jdbc:*` commands, or using the JDBC MBean.

Commands

`jdbc:ds-factories`

The `jdbc:ds-factories` command lists the available datasource factories, with the available driver.

For instance, once you installed the `jdbc` feature, you can install `pax-jdbc-postgresql` feature, providing the PostgreSQL datasource factory:

```
karaf@root()> feature:install pax-jdbc-postgresql
```

Then, you can see the factory available with `jdbc:ds-factories`:

```
karaf@root()> jdbc:ds-factories
Name                    | Class                   | Version | Registration bundle
------------------------|-------------------------|---------|-------------------------------
PostgreSQL JDBC Driver | org.postgresql.Driver | 42.2.8  | org.postgresql.jdbc42 [51]
```

You can see there the JDBC driver name and class that you can use in the `jdbc:ds-create` command.

NB: don't forget to install the `pax-jdbc-*` features for the database you want to connect:

```
pax-jdbc-db2
pax-jdbc-teradata
pax-jdbc-derby
pax-jdbc-derbyclient
pax-jdbc-h2
pax-jdbc-hsqldb
pax-jdbc-mariadb
pax-jdbc-oracle
pax-jdbc-postgresql
pax-jdbc-sqlite
pax-jdbc-mssql
pax-jdbc-jtds
```

`jdbc:ds-create`

The `jdbc:ds-create` command automatically creates a datasource definition file by leveraging pax-jdbc.

The `jdbc:ds-create` command requires either:

- the `--driverName` containing the JDBC driver name
- or the `--driverClass` containing the JDBC driver class name

Driver name and driver class can be found with `jdbc:ds-factories` command.

The `jdbc:ds-create` accepts a set of options and the name argument:

```
DESCRIPTION
        jdbc:ds-create

        Create a JDBC datasource config for pax-jdbc-config from a DataSourceFactory

SYNTAX
        jdbc:ds-create [options] name

ARGUMENTS
        name
                The JDBC datasource name

OPTIONS
        -dbName
                Database name to use
        --help
                Display this help message
        -dn, --driverName
                org.osgi.driver.name property of the DataSourceFactory
        -u, --username
                The database username
        -dc, --driverClass
                org.osgi.driver.class property  of the DataSourceFactory
        -p, --password
                The database password
        -url
                The JDBC URL to use
```

- the `name` argument is required. It's the name of the datasource. The name is used to identify the datasource, and to create the datasource definition file (`deploy/datasource-[name].xml`).
- the `-u` option is optional. It defines the database username.
- the `-url` option is optional. It defines the JDBC URL to access to the database.
- the `-p` option is optional. It defines the database password.

For instance, to create an embedded Apache Derby database in Apache Karaf, you can do:

```
karaf@root()> jdbc:ds-create -dn derby -url "jdbc:derby:test;create=true" test
```

We can see that this command created a configuration PID containing the datasource properties.

Another example using PostgreSQL driver class name (you can find with `jdbc:ds-factories` command):

```
karaf@root()> jdbc:ds-create -dc org.postgresql.Driver -url
"jdbc:postgresql://localhost:5432/test" --username user --password passwd test
```

or using datasource factory name (also provided by `jdbc:ds-factories` command):

```
karaf@root()> jdbc:ds-create -dn "PostgreSQL JDBC Driver" -url
"jdbc:postgresql://localhost:5432/test" --username user --password passwd test
```

`jdbc:ds-delete`

The `jdbc:ds-delete` command deletes a datasource.

```
karaf@root()> jdbc:ds-delete test
```

`jdbc:ds-list`

The `jdbc:ds-list` command lists the JDBC datasources:

```
karaf@root()> jdbc:ds-list
Name | Service Id | Product        | Version               | URL              | Status
-----|------------|----------------|-----------------------|------------------|-------
test | 112        | Apache Derby   | 10.8.2.2 - (1181258) | jdbc:derby:test | OK
```

`jdbc:ds-info`

The `jdbc:ds-info` command provides details about a JDBC datasource. The data source may be specified using name or service.id:

```
karaf@root()> jdbc:ds-info test
Property       | Value
-----------------------------------------------
driver.version | 10.8.2.2 - (1181258)
service.id     | 112
username       | APP
db.version     | 10.8.2.2 - (1181258)
db.product     | Apache Derby
driver.name    | Apache Derby Embedded JDBC Driver
url            | jdbc:derby:test
```

`jdbc:execute`

The `jdbc:execute` command executes a SQL query that doesn't return any result on a given JDBC datasource.

Typically, you can use the `jdbc:execute` command to create tables, insert values into tables, etc.

For instance, we can create a `person` table on our `test` datasource:

```
karaf@root()> jdbc:execute test "create table person(name varchar(100), nick
varchar(100))"
```

And we can insert some records in the `person` table:

```
karaf@root()> jdbc:execute test "insert into person(name, nick) values('foo','bar')"
karaf@root()> jdbc:execute test "insert into person(name, nick) values('test','test')"
```

`jdbc:query`

The `jdbc:query` command is similar to the `jdbc:execute` one but it displays the query result.

For instance, to display the content of the `person` table, we can do:

```
karaf@root()> jdbc:query test "select * from person"
NICK       | NAME
--------------------------------
bar        | foo
test       | test
```

`jdbc:tables`

The `jdbc:tables` command displays all tables available on a given JDBC datasource:

```
karaf@root()> jdbc:tables test
REF_GENERATION | TYPE_NAME | TABLE_NAME       | TYPE_CAT | REMARKS | TYPE_SCHEM |
TABLE_TYPE   | TABLE_SCHEM | TABLE_CAT | SELF_REFERENCING_COL_NAME
-------------------------------------------------------------------------------------------
-------------------------------------------------------------
               |           | SYSALIASES       |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSCHECKS        |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSCOLPERMS      |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSCOLUMNS       |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSCONGLOMERATES |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSCONSTRAINTS   |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSDEPENDS       |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSFILES         |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSFOREIGNKEYS   |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSKEYS          |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSPERMS         |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSROLES         |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSROUTINEPERMS  |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSSCHEMAS       |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSSEQUENCES     |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSSTATEMENTS    |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSSTATISTICS    |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSTABLEPERMS    |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSTABLES        |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSTRIGGERS      |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSVIEWS         |          |         |            | SYSTEM
TABLE | SYS         |           |
               |           | SYSDUMMY1        |          |         |            | SYSTEM
TABLE | SYSIBM      |           |
```

```
                |           | PERSON           |          |         |            |
TABLE         | APP         |           |
```

JMX JDBC MBean

The JMX JDBC MBean provides the JDBC datasources, and the operations to manipulate datasources and database.

The object name to use is `org.apache.karaf:type=jdbc,name=*`.

Attributes

The `Datasources` attribute provides a tabular data of all JDBC datasource, containing:

- `name` is the JDBC datasource name
- `service.id` is the JDBC datasource ID of OSGi service
- `product` is the database product backend
- `url` is the JDBC URL used by the datasource
- `version` is the database version backend.

Operations

- `create(name, type, jdbcDriverClassName, version, url, user, password, installBundles)` creates a JDBC datasource (the arguments correspond to the options of the `jdbc:create` command).
- `delete(name)` deletes a JDBC datasource.
- `info(datasource)` returns a Map (String/String) of details about a JDBC `datasource`.
- `tables(datasource)` returns a tabular data containing the tables available on a JDBC `datasource`.
- `execute(datasource, command` executes a SQL command on the given JDBC `datasource`.
- `query(datasource, query` executes a SQL query on the given JDBC `datasource` and return the execution result as tabular data.

4.17.6. JMS

The Apache Karaf MOM (Messaging Oriented Middleware/JMS) is an optional enterprise feature.

You have to install the `jms` feature first:

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

The `jms` feature doesn't install a JMS broker: it just installs the OSGi service, commands, and MBean to interact with a JMS broker (not the broker itself).

It means that you have to install a JMS broker itself.

This JMS broker can be available:

- outside of Apache Karaf, as a standalone broker. In that case, Apache Karaf JMS will remotely connect to the JMS broker. For instance, you can use this topology with Apache ActiveMQ or IBM WebsphereMQ.
- embedded in Apache Karaf. With this topology, Apache Karaf itself provides a JMS broker service. Apache ActiveMQ provides a native support in Apache Karaf.

For instance, you can install Apache ActiveMQ directly in Apache Karaf:

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

The `activemq-broker` feature installs:

- a Apache ActiveMQ broker directly in Apache Karaf, binding to the `61616` port number by default.
- the Apache ActiveMQ WebConsole bound to `http://0.0.0.0:8181/activemqweb` by default.

The Apache Karaf `jms` feature provides an OSGi service to create/delete JMS connection factories in the container and perform JMS operations (send or consume messages, get information about a JMS broker, list the destinations, ...).

This JMS OSGi service can be manipulated programmatically (see the developer guide for details), using the `jms:*` commands, or using the JMS MBean.

Commands

`jms:create`

The `jms:create` command creates a JMS connection factory in the Apache Karaf container. It automatically creates a blueprint XML file in the `deploy` folder containing the JMS connection factory definition corresponding to the type that you mentioned.

The `jms:create` command accepts different arguments and options:

```
karaf@root()> jms:create --help
DESCRIPTION
        jms:create

        Create a JMS connection factory.

SYNTAX
        jms:create [options] name

ARGUMENTS
        name
                The JMS connection factory name
                (required)

OPTIONS
        -t, --type
                The JMS connection factory type (ActiveMQ, Artemis or WebsphereMQ)
                (defaults to activemq)
        --pool
                The pool mechanism to use for this connection factory
                (defaults to pooledjms)
        --help
                Display this help message
        --url
                URL of the JMS broker. For WebsphereMQ type, the URL is hostname/port/
queuemanager/channel
                (defaults to tcp://localhost:61616)
        -p, --password
                Password to connect to the JMS broker
                (defaults to karaf)
        -u, --username
                Username to connect to the JMS broker
                (defaults to karaf)
```

- the `name` argument is required. It's the name of the JMS connection factory. The name is used to identify the connection factory, and to create the connection factory definition file (`deploy/connectionfactory-[name].xml`).

- the `-t` (`--type`) option is required. It's the type of the JMS connection factory. Currently only `activemq` and `webspheremq` types are supported. If you want to use another type of JMS connection factory, you can create the `deploy/connectionfactory-[name].xml` file by hand (using one as template).
- the `--url` option is required. It's the URL used by the JMS connection factory to connect to the broker. If the type is `activemq`, the URL looks like `tcp://localhost:61616`. If the type is `webspheremq`, the URL looks like `host/port/queuemanager/channel`.
- the `-u` (`--username`) option is optional (karaf by default). In the case of the broker requires authentication, it's the username used.
- the `-p` (`--password`) option is optional (karaf by default). In the case of the broker requires authentication, it's the password used.
- the `--pool` option is optional (`pooledjms` by default). It allows you to define the pooling framework you want to use: pooledjms, narayana, transx.

For instance, to create a JMS connection factory for a Apache ActiveMQ broker, you can do:

```
karaf@root()> feature:install pax-jms-activemq
karaf@root()> jms:create -t activemq --url tcp://localhost:61616 test
```

NOTE

When you use ActiveMQ, Karaf uses `transx` as connection factory pooling framework.

NOTE

The `jms:create` command doesn't install any feature or bundle providing the JMS connection factory classes (and dependencies). You have to install the required features (for instance `activemq-broker` feature for Apache ActiveMQ), or bundles (for IBM WebsphereMQ) providing the JMS connection factory packages and classes.

In the previous example, we assume that you previously installed the `activemq-broker` feature.

We can see the created JMS connection factory:

```
karaf@root()> la
...
151 | Active   |  80 | 0.0.0                | connectionfactory-test.xml
```

The `connectionfactory-test.xml` file has been created in the `deploy` folder.

By default, the `jms:create` command constructs a JNDI name for the connection factory: `/jms/[name]`.

It means that the connection factory name to use for the other `jms:*` commands is `/jms/[name]`.

`jms:delete`

The `jms:delete` command deletes a JMS connection factory. The `name` argument is the name that you used at creation time:

```
karaf@root()> jms:delete test
```

`jms:connectionfactories`

The `jms:connectionfactories` command lists the JMS connection factories:

```
karaf@root()> jms:connectionfactories
JMS Connection Factory
----------------------
/jms/test
```

`jms:info`

The `jms:info` command provides details about the JMS connection factory:

```
karaf@root()> jms:info /jms/test
Property | Value
-------------------
product  | ActiveMQ
version  | 5.9.0
```

You can see the JMS broker product and version.

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

`jms:queues`

The `jms:queues` command lists the JMS queues available on a JMS broker. For instance:

```
karaf@root()> jms:queues /jms/test
JMS Queues
----------
MyQueue
```

where `/jms/test` is the name of the JMS connection factory.

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

> **NOTE** Depending of the JMS connection factory type, this command may not work. For now, the command works only with Apache ActiveMQ.

`jms:topics`

The `jms:topics` command lists the JMS topics available on a JMS broker. For instance:

```
karaf@root()> jms:topics /jms/test
JMS Topics
----------
MyTopic
```

where `/jms/test` is the name of the JMS connection factory.

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

> **NOTE** Depending of the JMS connection factory type, this command may not work. For now, the command works only with Apache ActiveMQ.

`jms:send`

The `jms:send` command sends a message to a given JMS queue.

For instance, to send a message containing `Hello World` in the `MyQueue` queue, you can do:

```
karaf@root()> jms:send /jms/test MyQueue "Hello World"
```

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

`jms:consume`

The `jms:consume` command consumes messages from a JMS queue.

For instance, to consume all messages from `MyQueue`, you can do:

```
karaf@root()> jms:consume /jms/test MyQueue
2 message(s) consumed
```

If you want to consume only some messages, you can define a selector using the `-s` (`--selector`) option.

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

> **NOTE** The `jms:consume` command just consumes (so removes) messages from a JMS queue. It doesn't display the messages. If you want to see the details of messages, you can use the `jms:browse` command.

`jms:count`

The `jms:count` command counts the number of pending messages into a JMS queue.

For instance, if you want to know the number of messages on `MyQueue`, you can do:

```
karaf@root()> jms:count /jms/test MyQueue
Messages Count
--------------
8
```

If the JMS broker requires authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

`jms:browse`

The `jms:browse` command browses a JMS queue and display details about messages.

For instance, to browse the `MyQueue` queue:

```
karaf@root()> jms:browse /jms/test MyQueue
Message ID                              | Content        | Charset | Type | Correlation
ID | Delivery Mode | Destination     | Expiration | Priority | Redelivered | ReplyTo |
Timestamp
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------
ID:vostro-59602-1387462183019-3:1:1:1:1 | Hello World    | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:12 CET 2013
ID:vostro-59602-1387462183019-3:2:1:1:1 | Hello ActiveMQ | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:16 CET 2013
ID:vostro-59602-1387462183019-3:3:1:1:1 | Hello Karaf    | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:19 CET 2013
```

By default, the messages properties are not displayed. You can use the `-v` (`--verbose`) option to display the properties:

```
karaf@root()> jms:browse -v /jms/test MyQueue
Message ID                              | Content        | Charset | Type | Correlation
ID | Delivery Mode | Destination     | Expiration | Priority | Redelivered | ReplyTo |
Timestamp                    | Properties
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-----------------------------------
ID:vostro-59602-1387462183019-3:1:1:1:1 | Hello World    | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:12 CET 2013 |
ID:vostro-59602-1387462183019-3:2:1:1:1 | Hello ActiveMQ | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:16 CET 2013 |
ID:vostro-59602-1387462183019-3:3:1:1:1 | Hello Karaf    | UTF-8   |
|                 | Persistent    | queue://MyQueue | Never      | 4        | false
|         | Thu Dec 19 15:10:19 CET 2013 |
```

If you want to browse only some messages, you can define a selector using the `-s` (`--selector`) option.

If the JMS broker requires an authentication, you can use the `-u` (`--username`) and `-p` (`--password`) options.

`jms:move`

The `jms:move` command consumes all messages from a JMS queue and send it to another one.

For instance, to move all messages from `MyQueue` queue to `AnotherQueue` queue, you can do:

```
karaf@root()> jms:move /jms/test MyQueue AnotherQueue
3 message(s) moved
```

JMX JMS MBean

The JMX JMS MBean provides the attributes and operations to manipulate the JMS connection factories and JMS messages.

The object name to use is `org.apache.karaf:type=jms,name=*`.

Attributes

The `Connectionfactories` attribute provides the list of all JMS connection factories names.

Operations

- `create(name, type, url)` creates a JMS connection factory.
- `delete(name)` deletes a JMS connection factory.
- `Map<String, String> info(connectionFactory, username, password)` gets details about a JMS connection factory and broker.
- `int count(connectionFactory, queue, username, password)` counts the number of pending messages on a JMS queue.
- `List<String> queues(connectionFactory, username, password)` lists the JMS queues available on the JMS broker.
- `List<String> topics(connectionFactory, username, password)` lists the JMS topics available on the JMS broker.
- `TabularData browse(connectionFactory, queue, selector, username, password)` browses a JMS queue and provides a table of JMS messages.
- `send(connectionFactory, queue, content, replyTo, username, password)` sends a JMS message to a target queue.
- `int consume(connectionFactory, queue, selector, username, password)` consumes JMS messages from a JMS queue.

- `int move(connectionFactory, source, destination, selector, username, password)` moves messages from a JMS queue to another.

4.17.7. Persistence (JPA)

Apache Karaf provides JPA persistence providers (such as Apache OpenJPA) to be easy to use (in a OSGi way) and provide container managed persistence for applications (using Blueprint).

Apache Karaf embeds Aries JPA, providing a very easy way to develop applications that use JPA persistence.

See the developer guide for details about developing applications that use JPA.

Persistence engine features

Apache Karaf provides a set of ready to use persistence engine features:

- Apache OpenJPA. The `openjpa` feature installs the `jpa` feature with the Apache OpenJPA as persistence engine:

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

- Hibernate. The `hibernate` feature installs the `jpa` feature with the Hibernate persistence engine:

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

- EclipseLink. The `eclipselink` feature installs the `jpa` featue with the ElipseLink persistence engine:

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

4.17.8. EJB

This section describes how to add support of EJB in Apache Karaf. It doesn’t describe how to develop EJB applications.

Apache OpenEJB

Apache Karaf doesn’t provide "native" support of EJB (Enterprise Java Beans).

Apache OpenEJB provides EJB support for Apache Karaf by providing a set of features.

You have to update some Karaf configuration to have full OpenEJB support.

First, in the `etc/system.properties`, you have to append the following properties:

```
...
#
# OpenEJB scanner
#
openejb.deployments.classpath.exclude=bundle:*
openejb.deployments.classpath.filter.descriptors=true
```

Due to some OpenEJB version constraints, you also have to update the `etc/jre.properties` by changing the version of the `javax.xml.namespace` package, and remove the version of the `javax.annotation` package (provided by Geronimo Annotation API spec bundle, used by OpenEJB):

```
...
javax.annotation, \
javax.annotation.processing, \
...
javax.xml.namespace;version="1.0.0", \
...
```

It enables the OpenEJB bundles scanning, looking for EJBs.

After starting/restart Karaf to take these changes, we can install the OpenEJB feature:

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

By default, the `feature:repo-add openejb` command will install the latest OpenEJB version available.

You can specify a target version using the `version` argument:

```
karaf@root()> feature:repo-add openejb 4.5.2
```

Now, you have a set of new OpenEJB features available in your Apache Karaf container:

```
karaf@root()> la
...
openejb-core                    | 4.5.2 |           | openejb-features          |
openejb-server                  | 4.5.2 |           | openejb-features          |
openejb-cxf                     | 4.5.2 |           | openejb-features          |
openejb-rest                    | 4.5.2 |           | openejb-features          |
openejb-soap                    | 4.5.2 |           | openejb-features          |
```

You can add EJB support installing the `openejb-core` feature:

```
karaf@root()> feature:install openejb-core
```

Apache KarafEE

A custom distribution of Apache Karaf embedding OpenEJB is available in the Apache TomEE project.

The name of this custom distribution is KarafEE: https://svn.apache.org/repos/asf/tomee/karafee/

However, this project is now "deprecated", and all resources from KarafEE will move directly to Apache Karaf soon.

4.17.9. CDI

This section described how to add support of CDI, and embed a CDI container in Apache Karaf. It doesn't describe how to develop CDI applications. See the developer guide for that.

Pax CDI

Apache Karaf supports different CDI containers by using Pax CDI.

Pax CDI is pre-loaded in Apache Karaf.

You can see now a set of new CDI features available:

```
karaf@root()> feature:list|grep -i cdi
pax-cdi                                 | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI support
pax-cdi-1.1                             | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support
pax-cdi-1.2                             | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support
pax-cdi-weld                            | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI support
pax-cdi-1.1-weld                        | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support
pax-cdi-1.2-weld                        | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.2 support
pax-cdi-openwebbeans                    | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | OpenWebBeans CDI support
pax-cdi-web                             | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI support
pax-cdi-1.1-web                         | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support
pax-cdi-1.2-web                         | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.2 support
pax-cdi-web-weld                        | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI support
pax-cdi-1.1-web-weld                    | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.1 support
pax-cdi-1.2-web-weld                    | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.2 support
pax-cdi-web-openwebbeans                | 0.12.0                              |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | OpenWebBeans Web CDI support
deltaspike-core                         | 1.2.1                               |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike core support
deltaspike-jpa                          | 1.2.1                               |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike jpa support
deltaspike-partial-bean                 | 1.2.1                               |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike partial bean support
deltaspike-data                         | 1.2.1                               |          |
Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike data support
```

CDI Containers

Thanks to Pax CDI, Apache Karaf supports multiple CDI implementation versions, and different CDI containers.

You just have to install the feature corresponding to the CDI container and version that you want to use.

Apache OpenWebBeans

Apache Karaf provides a ready to use feature for Apache OpenWebBeans.

The `openwebbeans` feature automatically install the Pax CDI features and the Apache OpenWebBeans bundles:

```
karaf@root()> feature:install http
karaf@root()> feature:install pax-cdi-openwebbeans
```

JBoss Weld CDI container

Apache Karaf provides a ready to use feature for JBoss Weld.

The `weld` feature automatically install the Pax CDI features and the JBoss Weld bundles:

```
karaf@root()> feature:install pax-cdi-weld
```

4.17.10. HA/failover and cluster

Apache Karaf natively provides a failover mechanism. It uses a kind of master/slave topology where one instance is active and the others are in standby.

If you are looking for cluster of Apache Karaf instances (active/active), Apache Karaf Cellar (https://karaf.apache.org/projects.html) is a solution.

Karaf provides failover capability using either a simple lock file 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.

HA/failover (active/passive)

The Apache Karaf failover capability uses a lock system.

This container-level lock system allows bundles installed on the master to be preloaded on the slave, in order to provide faster failover performance.

Two types of lock are supported:

- filesystem lock
- database lock

When a first instance starts, if the lock is available, it takes the lock and become the master. If a second instance starts, it tries to acquire the lock. As the lock is already hold by the master, the instance becomes a slave, in standby mode (not active). A slave periodically check if the lock has been released or not.

Filesystem lock

The Apache Karaf instances share a lock on the filesystem. It means that the filesystem storing the lock has to be accessible to the different instances (using SAN, NFS, ...).

The configuration of the lock system has to be defined in the `etc/system.properties` file, on each instance (master/slave):

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

- `karaf.lock` property enables the the HA/failover mechanism
- `karaf.lock.class` property contains the class name providing the lock implementation. Here, we use the filesystem lock.
- `karaf.lock.dir` property contains the location where the lock will be written. All instances have to share the same lock.
- `karaf.lock.delay` property is the interval period (in milliseconds) to check if the lock has been released or not.

Database lock

It's not always possible and easy to have a shared filesystem between multiple Apache Karaf instances.

Instead of sharing a filesystem, Apache Karaf supports sharing a database.

The master instance holds the lock by locking a table in the database. If the master loses the lock, a waiting slave gains access to the locking table, acquire the lock on the table and starts.

The database lock uses JDBC (Java DataBase Connectivity). To use database locking, you have to:

- copy the JDBC driver in the `lib/ext` folder on each instance. The JDBC driver to use is the one corresponding to the database used for the locking system.

- update `etc/system.properties` file on each instance:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.lock.DefaultJDBCLock
karaf.lock.level=50
karaf.lock.delay=10000
karaf.lock.jdbc.url=jdbc:derby://dbserver:1527/sample
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
karaf.lock.lostThreshold=0
```

- `karaf.lock` property enabled the HA/failover mechanism
- `karaf.lock.class` property contains the class name providing the lock implementation. The `org.apache.karaf.main.lock.DefaultJDBCLock` is the most generic database lock system implementation. Apache Karaf supports lock systems for specific databases (see later for details).
- `karaf.lock.level` property is the container-level locking (see later for details).
- `karaf.lock.delay` property is the interval period (in milliseconds) to check if the lock has been released or not.
- `karaf.lock.lostThreshold` property is the count of attempts to re-acquire the lock before shutting down.
- `karaf.lock.jdbc.url` property contains the JDBC URL to the database (derby in this example).
- `karaf.lock.jdbc.driver` property contains the class name of the JDBC driver to use (derby in this example).
- `karaf.lock.jdbc.user` property contains the username to use to connect to the database.
- `karaf.lock.jdbc.password` property contains the password to use to connet to the database.
- `karaf.lock.jdbc.table` property contains the database table to use for the lock. Karaf will first try to find the table as specified in this property, and if not found, it will try the table name in lower and upper case.

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

NOTE | The `sample` database will be created automatically if it does not exist.

NOTE | If the connection to the database is lost, the master instance tries to gracefully shutdown, allowing a slave instance to become the master when the database is back. The former master instance will require a manual restart.

Lock on Oracle

Apache Karaf supports Oracle database for locking. The lock implementation class name to use is `org.apache.karaf.main.lock.OracleJDBCLock`:

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

The Oracle JDBC driver file (`ojdbc*.jar`) has to be copied in the `lib/ext` folder.

NOTE | The `karaf.lock.jdbc.url` property contains a JDBC URL which requires an active SID. It means that you must manually create the Oracle database instance first before using the lock mechanism.

Lock on Derby

Apache Karaf supports Apache Derby database for locking. The lock implementation class name to use is `org.apache.karaf.main.lock.DerbyJDBCLock`:

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

The Derby JDBC driver file name has to be copied in the `lib/ext` folder.

Lock on MySQL

Apache Karaf supports MySQL database for locking. The lock implementation class name to use is `org.apache.karaf.main.lock.MySQLJDBCLock`:

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

The MySQL JDBC driver file name has to be copied in `lib/ext` folder.

Lock on PostgreSQL

Apache Karaf supports PostgreSQL database for locking. The lock implementation class name to use is `org.apache.karaf.main.lock.PostgreSQLJDBCLock`:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.lock.PostgreSQLJDBCLock
karaf.lock.jdbc.url=jdbc:postgresql://127.0.0.1:1527/dbname
karaf.lock.jdbc.driver=org.postgresql.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=0
```

The PostgreSQL JDBC driver file has to be copied in the `lib/ext` folder.

Lock on Microsoft SQLServer

Apache Karaf supports Microsoft SQLServer database for locking. The lock implementation class name to use is `org.apache.karaf.main.lock.SQLServerJDBCLock`:

```
karaf.lock=true
karaf.lock.class=org.apache.karaf.main.lock.SQLServerJDBCLock
karaf.lock.level=50
karaf.lock.delay=10000
karaf.lock.jdbc.url=jdbc:jtds:sqlserver://127.0.0.1;databaseName=sample
karaf.lock.jdbc.driver=net.sourceforge.jtds.jdbc.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

The JTDS JDBC driver file has to be copied in the `lib/ext` folder.

Container-level locking

Apache Karaf supports container-level locking. It allows bundles to be preloaded into the slave instance. Thanks to that, switching to a slave instance is very fast as the slave instance already contains all required bundles.

The container-level locking is supported in both filesystem and database lock mechanisms.

The container-level locking uses the `karaf.lock.level` property:

```
karaf.lock.level=50
```

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

As reminder, the bundles start levels are specified in `etc/startup.properties`, in the `url=level` format.

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

NOTE

Using *hot* standby means that the slave instances are running and bound to some ports. So, if you use master and slave instances on the same machine, you have to update the slave configuration to bind the services (ssh, JMX, etc) on different port numbers.

Cluster (active/active)

Apache Karaf doesn't natively support clustering. By cluster, we mean several active instances, synchronized with each other.

However, Apache Karaf Cellar (http://karaf.apache.org/index/subprojects/cellar.html) can be installed to provide cluster support.

4.18. Monitoring and Management using JMX

Apache Karaf provides a complete JMX layer.

You can remotely connect to a running Apache Karaf instance using any JMX client (like jconsole).

The Apache Karaf features provide a set of MBeans, dedicating to monitoring and management.

4.18.1. Connecting

Apache Karaf exposes a complete MBean server that you can use remotely.

The default port number is 1099.

The JMX URL to use by default is:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
```

You have to provide an username and password to access the JMX layer. The JMX layer uses the security framework, and so, by default, it uses the users defined in `etc/users.properties`.

You can change the port numbers of the JMX layer in the `etc/org.apache.karaf.management.cfg` configuration file.

NOTE

For security reason, by default, `karaf` user is disabled. To allow the logon, you have to have an user. You can enable `karaf` user by uncommenting the user in `etc/users.properties`:

```
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

4.18.2. Configuration

The Apache Karaf JMX management layer is configured in the `etc/org.apache.karaf.management.cfg` configuration file:

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

#
# The properties in this file define the configuration of Apache Karaf's JMX Management
#

#
# Port number for RMI registry connection
#
rmiRegistryPort = 1099

#
# Port number for RMI server connection
#
rmiServerPort = 44444

#
# Name of the JAAS realm used for authentication
#
jmxRealm = karaf

#
# The service URL for the JMXConnectorServer
#
serviceUrl = service:jmx:rmi://0.0.0.0:${rmiServerPort}/jndi/
rmi://0.0.0.0:${rmiRegistryPort}/karaf-${karaf.name}

#
# JMXMP connector enabled
# NB: if you enable JMXMP, don't forget to add opendmk_jmxremote_optional jar in lib/boot
folder
#
```

```
jmxmpEnabled = false

#
# JMXMP connector host name
#
jmxmpHost = 127.0.0.1

#
# JMXMP connector port number
#
jmxmpPort = 9999

#
# JMXMP connector service URL
#
jmxmpServiceUrl = service:jmx:jmxmp://${jmxmpHost}:${jmxmpPort}

#
# Whether any threads started for the JMXConnectorServer should be started as daemon
threads
#
daemon = true

#
# Whether the JMXConnectorServer should be started in a separate thread
#
threaded = true

#
# The ObjectName used to register the JMXConnectorServer
#
objectName = connector:name=rmi

#
# The ObjectName used to register the JMXMP connector
#
jmxmpObjectName = connector:name=jmxmp

#
# Role name used for JMX access authorization
#
# jmxRole=admin
```

- `rmiRegistryPort` property contains the port number of the JMX RMI registry. Default is `1099`.
- `rmiServerPort` property contains the port number of the JMX RMI server. Default is `44444`.

- `jmxRealm` is the security realm to use as authentication backend. By default it uses the `karaf` realm.

By default, Karaf exposes JMX using RMI.

You can also enable JMXMP connector by settings `jmxmpEnabled=true`. You can then configure the JMXMP connector using the corresponding `jmxmp*` properties. In order to use JMXMP, you have to add `opendmk_jmxremote_optional` jar in `lib/boot` folder:

```
$ cd lib/boot
$ curl -O https://repo1.maven.org/maven2/org/glassfish/external/
opendmk_jmxremote_optional_jar/1.0-b01-ea/opendmk_jmxremote_optional_jar-1.0-b01-ea.jar
$ cd ../../bin
$ ./karaf
```

4.18.3. MBeans

Apache Karaf provides a bunch of MBeans.

The MBeans object names have the same format:

```
org.apache.karaf:type=[feature],name=[instance]
```

Installing additional Apache Karaf features and external applications can provide new MBeans.

The following MBeans list is non exhaustive:

- `org.apache.karaf:type=bundle,name=*`: management of the OSGi bundles.
- `org.apache.karaf:type=config,name=*`: management of the configurations.
- `org.apache.karaf:type=diagnostic,name=*`: creation of dumps containing the current Apache Karaf activity (used for diagnostic).
- `org.apache.karaf:type=feature,name=*`: management of the Apache Karaf features.
- `org.apache.karaf:type=http,name=*`: management of the HTTP service (provided by the `http` feature).
- `org.apache.karaf:type=instance,name=*`: management of the instances.
- `org.apache.karaf:type=jdbc,name=*`: management of the JDBC service (provided by the `jdbc` feature).

- `org.apache.karaf:type=jms,name=*`: management of the JMS service (provided by the `jms` feature).
- `org.apache.karaf:type=jndi,name=*`: management of the JNDI service (provided by the `jndi` feature).
- `org.apache.karaf:type=kar,name=*`: management of the KAR file.
- `org.apache.karaf:type=log,name=*`: management of the log service.
- `org.apache.karaf:type=obr,name=*`: management of the OBR service (provided by the `obr` feature).
- `org.apache.karaf:type=package,name=*`: details about packages exported/imported.
- `org.apache.karaf:type=service,name=*`: management of the OSGi services.
- `org.apache.karaf:type=system,name=*`: management of the Apache Karaf container itself (halt, restart, etc).
- `org.apache.karaf:type=web,name=*`: management of WebApplications (provided by the `war` feature).
- `org.apache.karaf:type=wrapper,name=*`: management of the service wrapper (provided by the `wrapper` feature).

4.18.4. RBAC

Apache Karaf provides a complete Role-Based Access Control to the JMX MBeans and operations.

Whenever a JMX operation is invoked, the roles of the user are checked against the required roles for this operation.

The access lists are defined in configuration file in the `etc` folder.

The relevant configuration is prefixed with `jmx.acl` and based on the JMX ObjectName that it applies to.

For instance, specific configuration for a MBean with the object name `foo.bar:type=Test` can be placed in the `etc/jmx.acl.foo.bar.Test.cfg` configuration file.

More generic configurations can be placed in the domain (e.g. jmx.acl.foo.bar.cfg) or at the top level (jmx.acl.cfg).

A simple configuration file looks like:

```
    # operation = role
    test = admin
    getVal = manager, viewer
```

Apache Karaf looks for required roles using the following process . The most specific configuration file is tried first. It means that in the previous example, the `etc/jmx.acl.foo.bar.Test.cfg` is looked at first. In this configuration, Apache Karaf looks for a:

1. Specific match for the invocation, e.g. `test(int)["17"] = role1`
2. Regex match for the invocation, e.g. `test(int)[/[0-9]/] = role2` In both cases, the passed argument is converted to a String for the comparison. If any of the above match, the search stops and the associated roles are used.
3. Signature match for the invocation, e.g. `test(int) = role3` If matched, the search stops and the associated roles are used.
4. Method name match for the invocation, e.g. `test = role4` If matched, the search stops and the associated roles are used.
5. A method name wildcard match, e.g. `te* = role5` For all the wildcard matches found in the current configuration file, the roles associated with the longest match are used. So if you have te* and * and the method invoked is *test*, then the roles defined with te* are used, not the ones defined with *.

If no matching definition is found, the most specific definition always takes the precedence.

You can find some configuration examples:

- Only a `manager` can call GC on the Memory MBean:

```
# etc/jmx.acl.java.lang.Memory.cfg
    gc = manager
```

- Bundles with ID between 0 and 49 can be stopped only by an `admin`, other bundles can be stopped by a `manager`:

```
# etc/jmx.acl.org.apache.karaf.bundles.cfg
    stop(java.lang.String)[/([1-4])?([0-9]/] = admin
    stop = manager
```

The `etc/jmx.acl.cfg` configuration file contains the global configuration for the invocation on any MBean that doesn't have a specific configuration:

```
# etc/jmx.acl.cfg
    list* = viewer
    get* = viewer
    is* = viewer
    set* = admin
    * = admin
```

By default, all "read-only" operations (`list*` , `get*` , `is*`) can be performed by a `viewer` , whereas the "read-write" operations can be performed only by an `admin` .

The `org.apache.karaf:type=security,area=jmx` MBean can be used to check whether the current user can access a certain MBean or invoke a specific operation on it. This MBean can be used by management clients (monitoring tools, etc) to decide whether to show certain MBeans or operations to the end user.

4.18.5. JMX-HTTP bridge with Jolokia

It's not always easy to use a JMX client with the RMI protocol.

Some monitoring tools (Nagios, Zabbix, ...) are not native JMX clients.

But most of them can use HTTP.

More over, you may want to write your own application/web application. In that case, HTTP and JSON can be very interesting and easy to remotely manage Apache Karaf.

Jolokia (http://www.jolokia.org/) can be installed in Apache Karaf as a remote JMX-HTTP bridge.

Karaf provides a jolokia feature, ready to install (you just need a Http Service installed first):

```
karaf@root()> feature:install http
karaf@root()> feature:install jolokia
```

By default, Jolokia is listening on the port `8181` (see the WebContainer (JSP/Servlet) page for details about the HTTP configuration).

If you point a browser on http://localhost:8181/jolokia you will see a JSON output like:

```
{"timestamp":1421765829,"status":200,"request":{"type":"version"},"value":{"protocol":"7.2
","config":{"useRestrictorService":"false","canonicalNaming":"true","includeStackTrace":"t
rue","listenForHttpService":"true","historyMaxEntries":"10","agentId":"192.168.134.10-5922
-6eb8d517-osgi","debug":"false","realm":"karaf","serializeException":"false","agentContext
":"\/jolokia","agentType":"servlet","policyLocation":"classpath:\/jolokia-
access.xml","user":"karaf","debugMaxEntries":"100","authMode":"jaas","mimeType":"text\/pla
in"},"agent":"1.2.4-SNAPSHOT","info":{"product":"felix","vendor":"Apache","version":"4.4.1
"`}
```

You can manipulate the Apache Karaf JMX layer via HTTP and JSON, via system tools (like `curl`, `jmx4perl`, monitoring tools (supporting HTTP/JSON), or web applications.

For instance, you can send a JSON request to get details about the current Apache Karaf heap memory usage.

The format of the request is:

```
{
    "type":"read",
    "mbean":"java.lang:type=Memory",
    "attribute":"HeapMemoryUsage",
    "path":"used"
}
```

We can send this JSON request using `curl` and get the result:

```
curl -u karaf -d
"{\"type\":\"read\",\"mbean\":\"java.lang:type=Memory\",\"attribute\":\"HeapMemoryUsage\",
\"path\":\"used\"}" http://localhost:8181/jolokia/ && echo ""
Enter host password for user 'karaf':
{"timestamp":1421765948,"status":200,"request":{"mbean":"java.lang:type=Memory","path":"us
ed","attribute":"HeapMemoryUsage","type":"read"},"value":69121000}
```

You can find details on the Jolokia website (http://www.jolokia.org) and in the documentation (http://www.jolokia.org/reference/html/).

4.18.6. Apache Karaf Decanter

Apache Karaf Decanter provides a complete monitoring solution including data history, turnkey dashboards, SLA and alerting support.

4.19. WebConsole

Apache Karaf provides an optional WebConsole.

This WebConsole provides a graphical web GUI to see and manage your Apache Karaf container.

You can use the WebConsole to:

- manage Apache Karaf features
- manage OSGi bundles
- manage the instances
- manage the confgurations
- manage the log service

The WebConsole is extensible via a plugins system. Some applications can add new pages to the WebConsole. For instance, Apache Karaf Cellar provides additional pages to administrate cluster groups, nodes, etc.

4.19.1. Installation

To enable the Apache Karaf WebConsole, you just have to install the `webconsole` feature:

```
karaf@root()> feature:install http
karaf@root()> feature:install webconsole
```

NB: you have to install a Http Service first as requirement, either `http` or `felix-http` feature.

The `webconsole` feature automatically installs the `http` feature (see the [WebContainer section|webcontainer] for details).

4.19.2. Access

The Apache Karaf WebConsole uses the WebContainer port number (see the [WebContainer section|webcontainer] for details) with the `/system/console` context.

By default, the Apache Karaf WebContainer port number is `8181`.

It means that the Apache Karaf WebConsole is accessible on the following URL: http://localhost:8181/system/console

As the Apache Karaf WebConsole uses the security framework, an username and password will be prompted.

You have to enter an username/password from the `karaf` realm. By default, you can use `karaf`/`karaf`.

See the Security section for details.

NOTE

For security reason, by default, `karaf` user is disabled. To allow the logon, you have to have an user. You can enable `karaf` user by uncommenting the user in `etc/users.properties`:

```
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
```

NOTE

By default, only users with the `admin` role are allowed to logon to the Apache Karaf WebConsole. Right now, the WebConsole doesn't use RBAC system as we have for console commands, or MBeans.

You can change the security configuration of the webconsole in the `etc/org.apache.karaf.webconsole.cfg` configuration file:

```
# The JAAS realm name to use for authentication
realm=karaf

# The role required to access the WebConsole
role=admin
```

4.20. Scheduler

Apache Karaf provides an optional Scheduler which provides a Service Listener which listens for Runnable Services and schedules their execution, based on the service properties.

This Scheduler implementation uses the Quartz Scheduler library to understand cron-like expressions.

4.20.1. Installation

To enable the Apache Karaf Scheduler, you just have to install the `scheduler` feature:

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

The `scheduler` feature automatically installs the `scheduler` command group, too:

```
scheduler:list
scheduler:schedule-script
scheduler:schedule-command
scheduler:unschedule
scheduler:reschedule
```

4.20.2. Configuration

All jobs allow configuration using service properties:

Table 1. Scheduler properties

Property	Default	Description
Scheduler.PROPERTY_SCHEDULER_PERIOD	-	Defines the period for a job. The period is expressed in seconds. This property needs to be of type Long.
Scheduler.PROPERTY_SCHEDULER_TIMES	-1	Defines the number of times the job is executed. -1 means infinite.
Scheduler.PROPERTY_SCHEDULER_IMMEDIATE	false	Define if a periodically job should be scheduled immediate. Default is to not startup immediate, the job is started the first time after the period has expired. This property needs to be of type Boolean.

Property	Default	Description
Scheduler.PROPERTY_SCHEDULER_EXPRESSION	-	Define the cron expression for a job. Must be a Quartz compatible expression.
Scheduler.PROPERTY_SCHEDULER_CONCURRENT	-	Define if the job can be run concurrently.
Scheduler.PROPERTY_SCHEDULER_NAME	-	Define the job name.

4.20.3. Schedule a new Job using the Whitebox-Service paradigm (recommended)

This example uses Declarative Services to register a Service of Type "org.apache.karaf.scheduler.Job" so that it is recognized by the Scheduler Service.

Alternatively, jobs can be registered as type "Runnable" in a more API neutral way. In this case you won't get the "JobContext" information though.

```
@Component(immediate = true, property = {
        Scheduler.PROPERTY_SCHEDULER_EXPRESSION + "=0 0/10 * * * ?",
} )
public class SchedulerPing implements Job {

    @Override
    public void execute(JobContext context) {
        // ..
    }
}
```

This will register a Job with the WhiteboxHandler. You can verify that the job is registered:

```
karaf@root()> scheduler:list
Name                    | Schedule
————————————————————————|————————————————————
Registered Service.185 | cron(0 0/10 * * * ?)
```

The Karaf scheduler can also schedule `Runnable` service.

For instance, if you have the following bean:

```
@Component(immediate = true, property = {
    "scheduler.period:Long=60",
    "scheduler.concurrent:Boolean=false",
    "scheduler.name=PingJob"
  }
)
public class PingThread implements Runnable {

  @Override
  public void run() {
    // ..
  }

}
```

This will register a job for the thread (runnable):

```
karaf@root()> scheduler:list
Name                        | Schedule
----------------------------+-------------------------------------------------
PingJob.126                 | at(2017-11-22T15:37:17.103+01:00, -1, 10)
```

It's also possible to pass the number of execution of the scheduler job using the `scheduler.times` service property:

```
@Component(immediate = true, property = {
    "scheduler.period:Long=60",
    "scheduler.times:Integer=5",
    "scheduler.concurrent:Boolean=false",
    "scheduler.name=PingJob"
  }
)
public class PingThread implements Runnable {

  @Override
  public void run() {
    // ..
  }

}
```

4.20.4. Schedule a new Job using the Gogo Shell

```
karaf@root()> scheduler:schedule-script --help
DESCRIPTION
        scheduler:schedule

    Schedule a script execution

SYNTAX
        scheduler:schedule [options] script

ARGUMENTS
        script
                The script to schedule

OPTIONS
        --at
                Absolute date in ISO format (ex: 2014-05-13T13:56:45)
        --concurrent
                Should jobs run concurrently or not (defaults to false)
        --period
                Time during executions (in seconds)
        --times
                Number of times this job should be executed
                (defaults to -1)
        --cron
                The cron expression
        --help
                Display this help message
        --name
                Name of this job
```

4.20.5. Schedule a command

```
karaf@root()> scheduler:schedule-command --help
DESCRIPTION
        scheduler:schedule-command

        Schedule a command execution

SYNTAX
        scheduler:schedule-command [options] command

ARGUMENTS
        command
                The command to schedule
                (required)

OPTIONS
        --period
                Time during executions (in seconds)
        --at
                Absolute date in ISO format (ex: 2014-05-13T13:56:45)
        --help
                Display this help message
        --times
                Number of times this job should be executed
                (defaults to -1)
        --concurrent
                Should jobs run concurrently or not (defaults to false)
        --cron
                The cron expression
```

4.20.6. Schedule a new Job using the Scheduler Service

Recommendation: Before using this low level api for registering jobs, consider using the whitebox approach instead.

```
..
import org.apache.karaf.scheduler.Scheduler;

@Component
public class Demo {

  @Reference Scheduler scheduler;

  public void useScheduler()
  {
    schedule(new MyJob(), scheduler.EXPR("0 0/10 * * * ?"));
  }

  class MyJob implements Job {
    ..
  }

}
```

4.20.7. Update scheduling of an existing job

You can change the scheduling of an existing job using `scheduler:reschedule` command.

This command works as the schedule command (using the same `at`, `period`, `cron`, ... options) but taking the job name as argument (as given by the `scheduler:list` command).

4.20.8. Using shared jobs store

By the default, the Apache Karaf scheduler uses a memory storage for jobs. It's local to a single Karaf instance.

You can setup several Karaf instances scheduler to use a shared job storage. It's especially interesting when you have a farm or cluster of Karaf instances.

For instance, you can use a database as storage.

You can create the datasource to this database, using the regular Karaf `jdbc` commands. For instance, to setup a `DataSource` for a remote Derby database, you can do:

```
karaf@root()> feature:install jdbc
karaf@root()> feature:install pax-jdbc-derbyclient
karaf@root()> jdbc:ds-create -dn derbyclient -url jdbc:derby://host:1527/karaf_scheduler
scheduler
karaf@root()> feature:install jndi
```

You have to init the DataBase tables with the SQL scripts provided in Quartz distribution (in the `docs/dbTables` directory).

Then you can configure the Karaf Scheduler to use this job storage, updating the `etc/org.apache.karaf.scheduler.cfg` file like this:

```
#============================================================================
# Configure Karaf Scheduler Properties
#============================================================================
org.quartz.scheduler.instanceName=Karaf
org.quartz.scheduler.instanceId=AUTO

#============================================================================
# Configure ThreadPool
#============================================================================
org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount=30
org.quartz.threadPool.threadPriority=5


#============================================================================
# Configure JDBC DataSource
#============================================================================
org.quartz.dataSource.scheduler.jndiURL=osgi:service/
javax.sql.DataSource/(osgi.jndi.service.name=scheduler)

#============================================================================
# Configure JDBC JobStore
#============================================================================
org.quartz.jobStore.class=org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.dataSource=scheduler
org.quartz.jobStore.driverDelegateClass=org.quartz.impl.jdbcjobstore.StdJDBCDelegate
```

Then several Karaf instances scheduler will share the same JDBC job store and can work in a "clustered" way.

4.21. Tuning

4.21.1. Garbage Collection

Like any Java applications, Apache Karaf uses a JVM. An important feature of the JVM is the Garbage Collector.

Apache Karaf default configuration is sized for small to medium needs and to work on most machines.

That's why this default configuration may appear like "small".

By default, Apache Karaf uses:

- `-Xms128M`
- `-Xmx512M`
- `-XX:+UnlockDiagnosticVMOptions`

On Sun/Oracle JVM:

- the Perm size is the JVM default (for Java 7)

On IBM JVM and AIX system:

- `-Xverify:none`
- `-Xdump:heap`
- `-Xlp`

For any container, it's always difficult to predict the usage of the resources and the behaviour of the artifacts deployed.

Generally speaking, a good approach for tuning is to enable `-verbose:gc` and use tools like VisualVM to identify the potential memory leaks, and see the possible optimisation of the spaces and GC.

You can find introduction about GC here: [http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html].

Java 6

If you have enough resources available on the machine, a good configuration may be:

- `-server`
- `-Xmx1024M`
- `-XX:MaxPermSize=640M`
- `-XX:+UseConcMarkSweepGC`
- `-XX:+UseParNewGC`

- `-XX:+CMSClassUnloadingEnabled`

It will give more resources to Apache Karaf, and avoid some perm space saturation if you do a lot of bundles refresh.

See http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html for details about Java 6 tuning.

Java 7

Java 7 introduces a new GC algorithm: the GC1.

Depending of the use cases (and usage of the heap), the new GC1 algorithm can give good performance improvements:

- `-XX:+UseG1GC`
- `-XX:-UseAdaptiveSizePolicy`

See http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html for details about Java 7 GC1 tuning.

You can find a good article about Java 7 tuning here: http://java-is-the-new-c.blogspot.de/2013/07/tuning-and-benchmarking-java-7s-garbage.html

4.21.2. Threads

In a system under high load, the number of threads can be very large.

WebContainer

If you use the Apache Karaf WebContainer, the Jetty connectors create threads to handle the incoming HTTP requests.

The `etc/jetty.xml` configuration file allows you to tune the Jetty connector.

For instance, the following connector configuration:

```
    <Call name="addConnector">
        <Arg>
            <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
                <Set name="host">
                    <Property name="jetty.host" />
                </Set>
                <Set name="port">
                    <Property name="jetty.port" default="8181" />
                </Set>
                <Set name="maxIdleTime">300000</Set>
                <Set name="Acceptors">2</Set>
                <Set name="statsOn">false</Set>
                <Set name="confidentialPort">8443</Set>
                <Set name="lowResourcesConnections">20000</Set>
                <Set name="lowResourcesMaxIdleTime">5000</Set>
            </New>
        </Arg>
    </Call>
```

defines the following properties:

- `maxIdleTime` is the maximum inactivity time for a connection.
- `lowResourcesConnections` defines the number of connections. If the current number of connections is greater than this value, the status is "low on resources". In that case, a new connection timeout is applied: the `lowResourceMaxIdleTime`.
- `Acceptors` defines the number of threads for incoming connections.

Apache Camel

For instance, if you use Apache Camel inside Apache Karaf, Camel components can create a lot of threads.

Apache Camel use the notion of `threadPoolProfile` to control the threads creation.

For instance, the following Camel configuration defines a pool creation strategy:

```
<threadPoolProfile id="defaultThreadPoolProfile" defaultProfile="true"
                   poolSize="10" maxPoolSize="20" maxQueueSize="1000"
                   rejectedPolicy="CallerRuns"/>
```

See the http://camel.apache.org for details.

Apache CXF

Apache CXF uses workqueues to handle server request/response.

You may see a `etc/org.apache.cxf.workqueues-default.cfg` configuration file. It's the default configuration applied to all workqueues (a workqueue can be associated to a specific endpoint).

On a workqueue, you can define the following properties about the threads:

- `org.apache.cxf.workqueue.default.highWaterMark` defines the maximum number of threads.
- `org.apache.cxf.workqueue.default.lowWaterMark` defines the minimum number of threads.
- `org.apache.cxf.workqueue.default.initialSize` defines the initial number of threads.

See the http://cxf.apache.org for details.

4.21.3. System packages

The `etc/jre.properties` defines the packages directly provided by the JVM.

Most of the time, the default configuration in Apache Karaf is fine and works in most of the use cases.

However, some times, you may not want to use the packages provided by the JVM, but the same packages provided by a bundle.

For instance, the JAXB version provided by the JVM is "old", and you want to use new JAXB bundles.

In that case, you have to comment the packages in `etc/jre.properties` to avoid to be provided by the JVM and use the ones from the bundles.

5. Developer Guide

5.1. Developer commands

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

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

5.1.1. Dump

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

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

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

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

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

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

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

You can specify the file name of the zip archive:

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

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

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

5.1.2. Diagnostic

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

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

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

```
karaf@root()> bundle:diag
Apache ServiceMix :: Bundles :: avro-ipc (81)
---------------------------------------------
Status: Installed
Unsatisfied Requirements:
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=javax.servlet)(version>=2.5.0)(!(version>=3.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=javax.servlet.http)(version>=2.5.0)(!(version>=3.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.data)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.file)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.generic)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.io)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.reflect)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.specific)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.apache.avro.util)(version>=1.7.0)(!(version>=2.0.0)))
[81.0] osgi.wiring.package; (osgi.wiring.package=org.apache.velocity)
[81.0] osgi.wiring.package; (osgi.wiring.package=org.apache.velocity.app)
[81.0] osgi.wiring.package; (osgi.wiring.package=org.apache.velocity.context)
[81.0] osgi.wiring.package; (osgi.wiring.package=org.apache.velocity.exception)
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.bootstrap)(version>=3.4.0)(!(version>=4.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.buffer)(version>=3.4.0)(!(version>=4.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel)(version>=3.4.0)(!(version>=4.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel.group)(version>=3.4.0)(!(version>=4.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.channel.socket.nio)(version>=3.4.0)(!(version>=4.0.
0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.codec.frame)(version>=3.4.0)(!(version>=4.0
.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.codec.oneone)(version>=3.4.0)(!(version>=4.
0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.jboss.netty.handler.execution)(version>=3.4.0)(!(version>=4.0.0
)))
```

```
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty)(version>=6.1.0)(!(version>=7.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.bio)(version>=6.1.0)(!(version>=7.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.nio)(version>=6.1.0)(!(version>=7.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.jetty.servlet)(version>=6.1.0)(!(version>=7.0.0)))
[81.0] osgi.wiring.package;
(&(osgi.wiring.package=org.mortbay.resource)(version>=6.1.0)(!(version>=7.0.0)))
```

5.1.3. Dynamic import

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

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

The purpose of dynamic import is to allow a bundle to be wired up to packages that may not be known about in advance. When a class is requested, if it cannot be solved via the bundle's existing imports, the dynamic import allows other bundles to be considered for a wiring import to be added.

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

5.1.4. OSGi framework

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

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

5.1.5. Stack traces printout

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

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

```
karaf@root()> stack-traces-print
Printing of stacktraces set to true
karaf@root()> bundle:start
java.lang.RuntimeException: Access to system bundle 0 denied. You can override with -f
        at
org.apache.karaf.bundle.command.BundlesCommand.assertNoSystemBundles(BundlesCommand.java:5
7)
        at
org.apache.karaf.bundle.command.BundlesCommand.doExecute(BundlesCommand.java:48)
        at
org.apache.karaf.bundle.command.BundlesCommandWithConfirmation.doExecute(BundlesCommandWit
hConfirmation.java:41)
        at org.apache.karaf.shell.console.AbstractAction.execute(AbstractAction.java:33)
        at
org.apache.karaf.shell.console.OsgiCommandSupport.execute(OsgiCommandSupport.java:39)
        at
org.apache.karaf.shell.commands.basic.AbstractCommand.execute(AbstractCommand.java:33)
        at sun.reflect.GeneratedMethodAccessor30.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:601)
        at org.apache.aries.proxy.impl.ProxyHandler$1.invoke(ProxyHandler.java:54)
        at org.apache.aries.proxy.impl.ProxyHandler.invoke(ProxyHandler.java:119)
        at
org.apache.karaf.shell.console.commands.$BlueprintCommand14083304.execute(Unknown Source)
        at sun.reflect.GeneratedMethodAccessor30.invoke(Unknown Source)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:601)
        at org.apache.aries.proxy.impl.ProxyHandler$1.invoke(ProxyHandler.java:54)
        at org.apache.aries.proxy.impl.ProxyHandler.invoke(ProxyHandler.java:119)
        at
org.apache.karaf.shell.console.commands.$BlueprintCommand14083304.execute(Unknown Source)
        at org.apache.felix.gogo.runtime.CommandProxy.execute(CommandProxy.java:78)
        at org.apache.felix.gogo.runtime.Closure.executeCmd(Closure.java:477)
        at org.apache.felix.gogo.runtime.Closure.executeStatement(Closure.java:403)
        at org.apache.felix.gogo.runtime.Pipe.run(Pipe.java:108)
        at org.apache.felix.gogo.runtime.Closure.execute(Closure.java:183)
        at org.apache.felix.gogo.runtime.Closure.execute(Closure.java:120)
        at
org.apache.felix.gogo.runtime.CommandSessionImpl.execute(CommandSessionImpl.java:89)
        at
org.apache.karaf.shell.console.impl.jline.ConsoleImpl$DelegateSession.execute(ConsoleImpl.
java:497)
        at org.apache.karaf.shell.console.impl.jline.ConsoleImpl.run(ConsoleImpl.java:198)
        at java.lang.Thread.run(Thread.java:722)
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3.doRun(ConsoleFactoryServ
ice.java:118)
```

```
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3$1.run(ConsoleFactoryServ
ice.java:109)
        at java.security.AccessController.doPrivileged(Native Method)
        at org.apache.karaf.jaas.modules.JaasHelper.doAs(JaasHelper.java:47)
        at
org.apache.karaf.shell.console.impl.jline.ConsoleFactoryService$3.run(ConsoleFactoryServic
e.java:107)
karaf@root()> stack-traces-print false
Printing of stacktraces set to false
karaf@root()> bundle:start
Error executing command: Access to system bundle 0 denied. You can override with -f
```

5.1.6. Bundle tree

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

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

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

5.1.7. Watch

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

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

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

The following command:

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

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

5.2. Scripting

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

5.2.1. Assignation

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

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

You can also assign a value to session variables:

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

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

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

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

```
karaf@root()> ($.context bundle 1) location
mvn:org.apache.karaf.jaas/org.apache.karaf.jaas.modules/4.0.0
```

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

5.2.2. Expressions

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

Examples:

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

Mathematical Operators

Operator	Description
+	Additive operator
-	Subtraction operator
*	Multiplication operator
/	Division operator

Operator	Description
%	Remainder operator (Modulo)
^	Power operator

Boolean Operators

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

Supported Functions

Function	Description
NOT(*expression*)	Boolean negation, 1 (means true) if the expression is not zero
IF(*condition*,*value_if_true*,*value_if_false*)	Returns one value if the condition evaluates to true or the other if it evaluates to false
RANDOM()	Produces a random number between 0 and 1

Function	Description
MIN(*e1*,*e2*)	Returns the smaller of both expressions
MAX(*e1*,*e2*)	Returns the bigger of both expressions
ABS(*expression*)	Returns the absolute (non-negative) value of the expression
ROUND(*expression*,precision)	Rounds a value to a certain number of digits, uses the current rounding mode
FLOOR(*expression*)	Rounds the value down to the nearest integer
CEILING(*expression*)	Rounds the value up to the nearest integer
LOG(*expression*)	Returns the natural logarithm (base e) of an expression
SQRT(*expression*)	Returns the square root of an expression
SIN(*expression*)	Returns the trigonometric sine of an angle (in degrees)
COS(*expression*)	Returns the trigonometric cosine of an angle (in degrees)
TAN(*expression*)	Returns the trigonometric tangens of an angle (in degrees)
SINH(*expression*)	Returns the hyperbolic sine of a value
COSH(*expression*)	Returns the hyperbolic cosine of a value
TANH(*expression*)	Returns the hyperbolic tangens of a value
RAD(*expression*)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians

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

Functions names are case insensitive.

Supported Constants

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

5.2.3. List, maps, pipes and closures

Using [], you can define an array variable:

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

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

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

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

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

```
karaf@root()> ($.context bundles) | grep -i felix
    0|Active     |    0|org.apache.felix.framework (4.2.1)
   21|Active     |   11|org.apache.felix.fileinstall (3.2.6)
   43|Active     |   10|org.apache.felix.configadmin (1.6.0)
   51|Active     |   30|org.apache.felix.gogo.runtime (0.10.0)
```

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

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

5.2.4. Startup

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

5.2.5. Constants and variables

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

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

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

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

5.2.6. Built-in variables and commands

Apache Karaf console provides built-in variables that are very useful for scripting:

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

Apache Karaf console provides commands for scripting:

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

See the full list of `shell` commands for details.

5.2.7. Leveraging existing Java capabilities (via reflection)

Apache Karaf console supports loading and execution of Java classes.

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

```
karaf@root()> ($.context bundle) loadClass foo
Error executing command: foo not found by org.apache.karaf.shell.console [17]
karaf@root()> $karaf.lastException printStackTrace
java.lang.ClassNotFoundException: foo not found by org.apache.karaf.shell.console [17]
    at
org.apache.felix.framework.BundleWiringImpl.findClassOrResourceByDelegation(BundleWiringIm
pl.java:1460)
    at org.apache.felix.framework.BundleWiringImpl.access$400(BundleWiringImpl.java:72)
    at
org.apache.felix.framework.BundleWiringImpl$BundleClassLoader.loadClass(BundleWiringImpl.j
ava:1843)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
    at org.apache.felix.framework.Felix.loadBundleClass(Felix.java:1723)
    at org.apache.felix.framework.BundleImpl.loadClass(BundleImpl.java:926)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.apache.felix.gogo.runtime.Reflective.invoke(Reflective.java:137)
    at org.apache.felix.gogo.runtime.Closure.executeMethod(Closure.java:527)
    at org.apache.felix.gogo.runtime.Closure.executeStatement(Closure.java:403)
    at org.apache.felix.gogo.runtime.Pipe.run(Pipe.java:108)
    at org.apache.felix.gogo.runtime.Closure.execute(Closure.java:183)
    at org.apache.felix.gogo.runtime.Closure.execute(Closure.java:120)
    at
org.apache.felix.gogo.runtime.CommandSessionImpl.execute(CommandSessionImpl.java:89)
    at org.apache.karaf.shell.console.jline.Console.run(Console.java:166)
    at java.lang.Thread.run(Thread.java:680)
```

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

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

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

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

5.2.8. Examples

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

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

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

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

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

5.3. Programmatically connect

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

5.3.1. To the console

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

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

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

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

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

public class Main {

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

        SshClient client = null;
        try {
            client = SshClient.setUpDefaultClient();
            client.start();
            ConnectFuture future = client.connect(host, port);
            future.await();
            ClientSession session = future.getSession();
            session.authPassword(user, password);
            ClientChannel channel = session.createChannel("shell");
            channel.setIn(System.in);
            channel.setOut(System.out);
            channel.setErr(System.err);
            channel.open();
            channel.waitFor(ClientChannel.CLOSED, 0);
        } catch (Throwable t) {
            t.printStackTrace();
            System.exit(1);
        } finally {
            try {
                client.stop();
            } catch (Throwable t) { }
        }
        System.exit(0);
    }

}
```

5.3.2. To the management layer

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

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

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

```
javax.management.*;

public class Main {

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

}
```

5.4. Branding

5.4.1. Console

You can "brand" the Apache Karaf console.

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

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

There are 2 ways of branding the Karaf console:

1. adding a branding.properties file to etc
2. creating a branding bundle.

5.4.2. Adding a branding.properties file to etc

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

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

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

Start Karaf and you will see your branded Karaf console.

5.5. Adding a branding-ssh.properties file to etc

In a similar way, a `etc/branding-ssh.properties` file can be added. It should contain the same `welcome` and `prompt` entries but those will be used for external clients connecting through ssh. The values default to the ones in `etc/branding.properties` if this file is not defined.

5.5.1. Branding bundle

See https://github.com/apache/karaf/blob/master/examples/karaf-branding-example/README.md

5.5.2. WebConsole

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

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

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

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


webconsole.brand.name = My Web Console

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

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

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

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

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

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

    <modelVersion>4.0.0</modelVersion>

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-DocURL>http://felix.apache.org/site/apache-
karaf.html</Bundle-DocURL>
                        <Fragment-Host>org.apache.karaf.webconsole.console;bundle-
version="[3,4)"</Fragment-Host>
                        <Export-Package>!*</Export-Package>
                        <Import-Package>
                            javax.servlet;version=2.4,
                            javax.servlet.http;version=2.4,
                            !org.apache.felix.webconsole*,
                            org.apache.aries.blueprint,
                            org.osgi.service.blueprint.container,
                            org.osgi.service.blueprint.reflect,
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

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

5.6. Extending

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

5.6.1. Shell commands

See https://github.com/apache/karaf/blob/master/examples/karaf-command-example/README.md to add your own shell commands.

5.6.2. WebConsole

You can also extend the Apache Karaf WebConsole by providing and installing a webconsole plugin.

A plugin is an OSGi bundle that registers a Servlet as an OSGi service with webconsole properties.

5.7. Using the karaf-maven-plugin

The Karaf Maven plugin allows you:

- to work with Karaf features: verify and validate a features descriptor, add features bundle into a repository, create a KAR archive from a features descriptor, etc.
- to create Karaf commands help: it generates help from Karaf commands
- to modify Karaf instances and create distributions

See https://github.com/apache/karaf/blob/master/examples/karaf-maven-example/README.md for details.

5.7.1. Packaging

The most generally useful features of the karaf-maven-plugin are exposed as packagings. To use the packagings the pom or an ancestor must configure the karaf-maven-plugin with extensions:

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <version>${project.version}</version>
                <extensions>true</extensions>
            </plugin>
        </plugins>
    </pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.karaf.tooling</groupId>
            <artifactId>karaf-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Then specify the packaging in your project as usual, e.g.

```
<packaging>kar</packaging>
```

Packaging	Description
feature	The feature packaging verifies a features.xml descriptor using the `karaf:verify` goal.
kar	The kar packaging generates a features.xml descriptor using the `karaf:features-generate-descriptor` and then packages a kar using the `karaf:features-create-kar` goal.
karaf-assembly	Assembles a Karaf server based on the features descriptors and kar files listed as Maven dependencies.

5.7.2. Commands goals

The `karaf-maven-plugin` is able to generate documentation for Karaf commands

`karaf:commands-generate-help`

The `karaf:commands-generate-help` goal generates documentation containing Karaf commands help.

It looks for Karaf commands in the current project class loader and generates the help as displayed with the `--help` option in the Karaf shell console.

Example

The example below generates help for the commands in the current project:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${project.version}</version>

        <executions>
          <execution>
            <id>document-commands</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>commands-generate-help</goal>
            </goals>
            <configuration>
              <targetFolder>${project.build.directory}/docbook/sources</targetFolder>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`targetFolder`	`File`	The directory where the documentation output files are to be generated. Default value: ${project.build.directory}/docbkx/sources
`format`	`String`	The output format (docbx, asciidoc, or conf) of the commands documentation. Default value: docbx
`classLoader`	`String`	The class loader to use in loading the commands. Default value: ${project}

5.7.3. Features and kar goals

> **NOTE** You should use the features or kar packaging instead of these individual goals.

The `karaf-maven-plugin` provides several goals to help you create and verify features XML descriptors as well as leverage your features to create a custom Karaf distribution.

`karaf:features-generate-descriptor`

The `karaf:features-generate-descriptor` goal generates a features XML file based on the Maven dependencies. By default, it will follow Maven transitive dependencies, stopping when it encounters bundles already present in features that are Maven dependencies.

A record of the dependency tree search can be found in target/history/treeListing.txt.

You can track dependency changes and warn or fail on change.

Configuration

Specify the packaging as a top level element

```
<packaging>feature</packaging>
```

You can supply a feature descriptor to extend in `src/main/feature/feature.xml`.

Parameter Name	Type	Description
aggregateFeatures	boolean (false)	Specifies processing of feature repositories that are (transitive) Maven dependencies. If false, all features in these repositories become dependencies of the generated feature. If true, all features in these repositories are copied into the generated feature repository.
startLevel	int	The start level for the bundles determined from Maven dependencies. This can be overridden by specifying the bundle in the source feature.xml with the desired startlevel.
includeTransitiveDependency	boolean (true)	Whether to follow Maven transitive dependencies.
checkDependencyChange	boolean (false)	Whether to record dependencies in `src/main/history/dependencies.xml` for change tracking.

Parameter Name	Type	Description
warnOnDependencyChange	boolean (false)	whether to fail on changed dependencies (false, default) or warn in the build output (true).
logDependencyChanges	boolean (false)	If true, added and removed dependencies are shown in `target/history`.
overwriteChangedDependencies	boolean (false)	If true, the `src/main/history/dependencies.xml` file will be overwritten if it has changed.
markRuntimeScopeAsDependency	boolean (true)	Whether to mark dependencies' in the `runtime` scope with `dependency="true"`.
markTransitiveAsDependency	boolean (false)	Whether to mark transitive dependencies' with `dependency="true"`.

Example

```
<project>
...
  <packaging>feature</packaging>
  <dependencies>
    <dependency>
      <groupId>org.apache</groupId>
      <artifactId>bundle1</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${project.version}</version>
        <extensions>true</extensions>
        <configuration>
          <enableGeneration>true</enableGeneration>
        </configuration>
        <executions>
          <execution>
            <id>generate-features-file</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>features-generate-descriptor</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

`karaf:verify`

Except in unusual circumstances, use the `<packaging>feature</packaging>` to run this goal.

The `karaf:verify` goal verifies and validates a features XML descriptor by checking if all the required imports for the bundles defined in the features can be matched to a provided export.

By default, the plugin tries to add the Karaf core features (standard and enterprise) in the repositories set. It means that it's not required to explicitly define the Karaf features descriptor in the repository section of your features descriptor.

Example

The example below validates the features defined in the `target/features.xml` by checking all the imports and exports. It reads the definition for the packages that are exported by the system bundle from the `src/main/resources/config.properties` file.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <configuration>
        </configuration>
        <executions>
          <execution>
            <id>verify</id>
            <phase>process-resources</phase>
            <goals>
              <goal>verify</goal>
            </goals>
            <configuration>
              <descriptors>
                <descriptor>mvn:org.apache.karaf.features/framework/4.0.4/xml/
features</descriptor>
                <descriptor>file:${project.build.directory}/feature/
feature.xml</descriptor>
              </descriptors>
              <distribution>org.apache.karaf.features:framework</distribution>
              <javase>1.8</javase>
              <framework>
                <feature>framework</feature>
              </framework>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`descriptors`	`String[]`	The list of features XML repositories to use for the verify
`features`	`String[]`	The list of features to verify. If not specified, all features in the descriptors will be verified.
`framework`	`String[]`	The features providing the Karaf framework (optional)
`javase`	`String`	The Java version to use for the verify

`karaf:features-add-to-repository`

Consider using the karaf-assembly packaging which makes it easy to assemble a custom distribution in one step instead of this individual goal.

The `karaf:features-add-to-repository` goal adds all the required bundles for a given set of features into directory. You can use this goal to create a `/system` directory for building your own Karaf-based distribution.

By default, the Karaf core features descriptors (standard and enterprise) are automatically included in the descriptors set.

Example

The example below copies the bundles for the `spring` and `war` features defined in the Karaf features XML descriptor into the `target/features-repo` directory.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${project.version}</version>

        <executions>
          <execution>
            <id>features-add-to-repo</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>features-add-to-repository</goal>
            </goals>
            <configuration>
              <descriptors>
                <descriptor>mvn:org.apache.karaf.features/standard/4.0.0/xml/
features</descriptor>
                <descriptor>mvn:my.groupid/my.artifactid/1.0.0/xml/features</descriptor>
              </descriptors>
              <features>
                <feature>spring</feature>
                <feature>war</feature>
                <feature>my</feature>
              </features>
              <repository>target/features-repo</repository>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`descriptors`	`String[]`	List of features XML descriptors where the features are defined
`features`	`String[]`	List of features that bundles should be copied to the repository directory

Name	Type	Description
`repository`	`File`	The directory where the bundles will be copied by the plugin goal
`timestampedSnapshot`	`boolean`	For SNAPSHOT artifacts, if false we use the base version (foo-1.0-SNAPSHOT), else we use the timestamped version (foo-1.0-2019xxxx). Default value: false

`karaf:create-kar`

> **NOTE** Except in unusual circumstances, use the `<packaging>kar</packaging>` to run this goal.

The `karaf:kar` goal assembles a KAR archive from a features XML descriptor file, normally generated in the same project with the `karaf:features-generate-descriptor` goal.

There are two important directories in a kar:

- `repository/` contains a Maven structured repository of artifacts to be copied into the Karaf repository. The features descriptor and all the bundles mentioned in it are installed in this directory.
- `resources/` contains other resources to be copied over the Karaf installation.

Everything in `target/classes` is copied into the kar. Therefore resources you want installed into Karaf need to be in e.g. `src/main/resources/resources`. This choice is so other resources such as legal files from the maven-remote-resources-plugin can be included under META-INF in the kar, without getting installed into Karaf.

Example

```
<project>
...
  <packaging>kar</packaging>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${project.version}</version>
        <extensions>true</extensions>
        <!-- There is no useful configuration for the kar mojo. The features-generate-
descriptor mojo configuration may be useful -->
      </plugin>
    </plugins>
  </build>
</project>
```

`karaf:install-kar`

5.7.4. Instances and distributions goals

The `karaf-maven-plugin` helps you to build custom Karaf distributions or archives existing Karaf instances:

`karaf:assembly`

> **NOTE** This goal is run as part of the karaf-assembly packaging.

The `karaf:assembly` goal creates a Karaf instance (assembly) filesystem using the Maven dependencies and a provided configuration.

Example

```
...
<packaging>karaf-assembly</packaging>
...
<dependencies>
    <dependency>
        <groupId>org.apache.karaf.features</groupId>
        <artifactId>framework</artifactId>
        <type>kar</type>
    </dependency>
    <dependency>
        <groupId>org.apache.karaf.features</groupId>
        <artifactId>standard</artifactId>
        <classifier>features</classifier>
        <type>xml</type>
        <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.karaf.tooling</groupId>
            <artifactId>karaf-maven-plugin</artifactId>
            <extensions>true</extensions>
            <configuration>
                <finalName>${project.artifactId}</finalName>
                <installedBundles>
                    <installedBundle>${additional.bundle}</installedBundle>
                </installedBundles>
                <bootFeatures>
                    <feature>bundle</feature>
                    <feature>config</feature>
                    <feature>diagnostic</feature>
                    <feature>feature</feature>
                    <feature>jaas</feature>
                    <feature>shell</feature>
                    <feature>log</feature>
                    <feature>management</feature>
                    <feature>package</feature>
                    <feature>shell-compat</feature>
                    <feature>ssh</feature>
                    <feature>system</feature>
                    <feature>wrap</feature>
                </bootFeatures>
            </configuration>
        </plugin>
    </plugins>
</build>
```

By default, the generated Karaf instance is a dynamic distribution (it's started with default set of resources and then you can deploy new applications in this instance).

It's also possible to generate a Karaf instance as a static distribution (kind of immutable):

```
    <dependencies>
        <dependency>
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>static</artifactId>
            <type>kar</type>
        </dependency>
        <dependency>
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>standard</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
        </dependency>
        <dependency>
            <groupId>org.apache.karaf.services</groupId>
            <artifactId>org.apache.karaf.services.staticcm</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <startupFeatures>
                        <startupFeature>static-framework</startupFeature>
                        <startupFeature>scr</startupFeature>
                        <startupFeature>http-whiteboard</startupFeature>
                    </startupFeatures>
                    <framework>static</framework>
                    <useReferenceUrls>true</useReferenceUrls>
                    <environment>static</environment>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Parameters

Name	Type	Description
`sourceDirectory`	`File`	Base directory used to overwrite resources in generated assembly after the build (resource directory). Default value: ${project.basedir}/src/main/resources/assembly
`workDirectory`	`File`	Base directory used to copy the resources during the build (working directory). Default value: ${project.build.directory}/assembly
`featuresProcessing`	`File`	Optional location for custom features processing XML configuration (`etc/org.apache.karaf.features.cfg`). Default value: null
`installAllFeaturesByDefault`	`Boolean`	For given stage (startup, boot, install) if there are no stage-specific features and profiles, all features from stage-specific repositories will be used. Default value: true
`environment`	`String`	An environment identifier that may be used to select different variant of PID configuration file, e.g., `org.ops4j.pax.url.mvn.cfg#docker`. Default value: null
`framework`	`String[]`	The features providing the Karaf framework (optional)
`javase`	`String`	The Java version to use for the verify

Name	Type	Description
`defaultStartLevel`	`int`	Default start level for bundles in features that don't specify it. Default value: 30
`startupRepositories`	`List<String>`	List of compile-scope features XML files to be used in startup stage (etc/startup.properties).
`bootRepositories`	`List<String>`	List of runtime-scope features XML files to be used in boot stage (etc/org.apache.karaf.features.cfg)
`installedRepositories`	`List<String>`	List of provided-scope features XML files to be used in install stage
`blacklistedRepositories`	`List<String>`	List of blacklisted repository URIs. Blacklisted URI may use globs and version ranges.
`startupFeatures`	`List<String>`	List of features from compile-scope features XML files and KARs to be installed into system repo and listed in etc/startup.properties.
`bootFeatures`	`List<String>`	List of features from runtime-scope features XML files and KARs to be installed into system repo and listed in featuresBoot property in etc/org.apache.karaf.features.cfg
`installedFeatures`	`List<String>`	List of features from provided-scope features XML files and KARs to be installed into system repo and not mentioned elsewhere.

Name	Type	Description
`blacklistedFeatures`	`List<String>`	List of feature blacklisting clauses. Each clause is in one of the formats (feature-name, feature-name;range=version-or-range, feature-name/version-or-range
`startupBundles`	`List<String>`	List of compile-scope bundles added to etc/startup.properties
`bootBundles`	`List<String>`	List of runtime-scope bundles wrapped in special feature added to featuresBoot property in etc/org.apache.karaf.features.cfg
`installedBundles`	`List<String>`	List of provided-scope bundles added to system repo
`blacklistedBundles`	`List<String>`	List of blacklisted bundle URIs. Blacklisted URI may use globs and version ranges.
`profilesUris`	`List<String>`	List of profile URIs to use
`startupProfiles`	`List<String>`	List of profiles names to load from configured `profilesUris` and use as startup profiles.
`bootProfiles`	`List<String>`	List of profiles names to load from configured `profilesUris` and use as boot profiles.
`installedProfiles`	`List<String>`	List of profiles names to load from configured `profilesUris` and use as installed profiles.
`blacklistedProfiles`	`List<String>`	List of blacklisted profile names (possibly using * glob).

Name	Type	Description
`writeProfiles`	`boolean`	When assembly custom distribution, we can include generated and added profiles in the distribution itself, in `${karaf.etc}/profiles` directory. Default value: false
`generateConsistencyReport`	`String`	When assembly custom distribution, we can also generate an XML/XSLT report with the summary of bundles. This parameter specifies target directory, to which `bundle-report.xml` and `bundle-report-full.xml` (along with XSLT stylesheet) will be written.
`consistencyReportProjectName`	`String`	When generating consistency report, we can specify project name. Default value: Apache Karaf
`consistencyReportProjectVersion`	`String`	When generating consistency report, we can specify project version. Default value: ${project.version}

`karaf:archive`

NOTE | This goal is run as part of the karaf-assembly packaging.

The `karaf:archive` goal packages a Karaf instance archive from a given assembled instance.

Both tar.gz and zip formats are generated in the destination folder.

Example

The example below create archives for the given Karaf instance:

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${project.version}</version>
          <executions>
            <execution>
              <id>generate</id>
              <phase>package</phase>
              <goals>
                <goal>archive</goal>
              </goals>
              <configuration>
                <destDir>${project.build.directory}</destDir>

<targetServerDirectory>${project.build.directory}/assembly</targetServerDirectory>
                <targetFile>${project.file}</targetFile>
              </configuration>
            </execution>
          </executions>
        </plugin>
    </plugins>
  </build>
</project>
```

Parameters

Name	Type	Description
`destDir`	`File`	The target directory of the project. Default value: ${project.build.directory}
`targetServerDirectory`	`File`	The location of the server repository. Default value: ${project.build.directory}/assembly
`pathPrefix`	`String`	Path prefix of files in the created archive. Default value: ${project.artifactId}-${project.version}

Name	Type	Description
`usePathPrefix`	`boolean`	Use a path prefix of files in the created archive. Default value: false
`targetFile`	`File`	The target file to set as the project's artifact. Default value: ${project.artifactId}-${project.version}
`archiveZip`	`Boolean`	Switches creation of *.zip artifact on or off. Default value: true
`archiveTarGz`	`Boolean`	Switches creation of *.tar.gz artifact on or off. Default value: true
`attach`	`boolean`	Whether to attach the resulting assembly to the project as an artifact. Default value: true
`classifier`	`String`	If supplied, the classifier for the artifact when attached.
`useSymLinks`	`boolean`	Use symbolic links in tar.gz or zip archives. Symbolic links are not very well supported by windows Platform. At least, is does not work on WinXP + NTFS, so do not include them for now. Default value: false

5.7.5. Run, client, deploy goals

These goals allows you to run Karaf via Maven or interact with a running instance.

`karaf:run`

`karaf:run` goal runs a Karaf instance from Maven.

It can use the current project if it's a bundle to directly deploy the project artifact in the running Karaf instance.

Example

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>run</id>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <karafDistribution>mvn:org.apache.karaf/apache-karaf/4.2.4/
zip</karafDistribution>
                            <keepRunning>false</keepRunning>
                            <deployProjectArtifact>false</deployProjectArtifact>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

Parameters

Name	Type	Description
`karafDirectory`	`File`	Directory containing Karaf container base directory. Default value: ${project.build.directory}/karaf
`karafDistribution`	`File`	Location where to download the Karaf distribution. Default value: mvn:org.apache.karaf/apache-karaf/LATEST/zip

Name	Type	Description
`deployProjectArtifact`	`boolean`	Define if the project artifact should be deployed in the started container or not. Default value: true
`featureRepositories`	`String[]`	A list of URLs referencing feature repositories that will be added to the karaf instance started by this goal.
`featuresToInstall`	`String`	Comma-separated list of features to install.
`keepRunning`	`boolean`	Define if the Karaf container keep running or stop just after the goal execution. Default value: true
`startSsh`	`boolean`	Define if the Karaf embedded sshd should be started or not. Default value: false

`karaf:client`

The `karaf:client` interacts with a running Karaf instance directly from Maven via SSH.

Example

```
<execution>
    <id>client</id>
    <goals>
        <goal>client</goal>
    <goals>
    <configuration>
        <commands>
            <command>
                <rank>1</rank>
                <command>feature:install foo</command>
            </command>
        </commands>
    </configuration>
</execution>
```

Parameters

Name	Type	Description
`port`	`int`	SSH port of the running Karaf instance. Default value: 8101
`host`	`String`	Hostname or IP address of the running Karaf instance. Default value: localhost
`user`	`String`	The username to connect to the running Karaf instance. Default value: karaf
`password`	`String`	The password to connect to the running Karaf instance. Default value: karaf
`keyFile`	`File`	The key file to use to connect to the running Karaf instance.

Name	Type	Description
`attempts`	`int`	The number of attempts to connect to the running Karaf instance. Default value: 0
`delay`	`int`	The delay (in second) to wait before trying to connect to the Karaf instance. Default value: 2
`commands`	`List<CommandDescriptor>`	The list of the commands to execute on the running Karaf instance.
`scripts`	`List<ScriptDescriptor>`	The list of the scripts to execute on the running Karaf instance.
`skip`	`boolean`	Skip or not the execution of the client goal execution. Default value: false

`karaf:deploy`

The `karaf:deploy` goal allows you to deploy bundles on a running Karaf instance (via SSH or JMX).

Example

```
<execution>
    <id>deploy</id>
    <goals>
        <goal>deploy</goal>
    <goals>
    <configuration>
        <useProjectArtifact>true</useProjectArtifact>
    </configuration>
</execution>
```

Parameters

Name	Type	Description
`port`	`int`	SSH port of the running Karaf instance. Default value: 8101
`host`	`String`	Hostname or IP address of the running Karaf instance. Default value: localhost
`user`	`String`	The username to connect to the running Karaf instance. Default value: karaf
`password`	`String`	The password to connect to the running Karaf instance. Default value: karaf
`keyFile`	`File`	The key file to use to connect to the running Karaf instance.
`attempts`	`int`	The number of attempts to connect to the running Karaf instance. Default value: 0
`delay`	`int`	The delay (in second) to wait before trying to connect to the Karaf instance. Default value: 2
`instance`	`String`	Name of the Karaf instance. Default value: karaf-root
`useSsh`	`boolean`	Define if the deployment is done via SSH or via JMX. Default value: false

Name	Type	Description
`useProjectArtifact`	`boolean`	Define if deploy uses the current Maven project artifact (in the case of the project is a bundle project). Default value: true
`artifactLocations`	`List<String>`	The list of bundle locations (mvn, http, file, ... URLs) to deploy.

5.7.6. Dockerfile, Docker

The Karaf Maven plugin is also able to generate Dockerfile and eventually interact with a local Docker daemon.

Dockerfile

The `karaf:dockerfile` goal creates a ready to use Dockerfile for a given Karaf distribution/assembly.

Example

```
<execution>
    <id>dockerfile</id>
    <goals>
        <goal>dockerfile</goal>
    </goals>
</execution>
```

Parameters

Name	Type	Description
`destDir`	`File`	The directory where the Dockerfile is generated. Default value: ${project.build.directory}

Name	Type	Description
`assembly`	`File`	The directory containing the Karaf assembly (as generated by `karaf:assembly`). Default value: ${project.build.directory}/assembly

Docker

The `karaf:docker` goal interacts directly with a local Docker daemon (via the command line) to directly create a Docker image with your Karaf assembly.

> **NOTE** This goal requires a local Docker daemon and runs only on Unix. The `docker` command has to be in the `PATH`.

Example

```
<execution>
    <id>docker</id>
    <goals>
        <goal>docker</docker>
    </goals>
</execution>
```

Parameters

Name	Type	Description
`location`	`File`	The directory containing the Dockerfile (can be generated by `karaf:dockerfile`). Default value: ${project.build.directory}
`imageName`	`String`	The name of the generated Docker image. Default value: karaf

5.8. Custom distributions

As Karaf is an OSGi container, it's heavily used as as application and middleware kernel.

You may wish to construct your own Karaf distribution preconfigured to your requirements.

This custom distribution could contain:

- branding to change the Karaf console look-and-feel
- configuration files (in the etc folder) altered to your requirements
- pre-packaged artifacts in the deploy folder
- a pre-populated system repository (containing your own bundle and features descriptor)
- renamed or specific scripts in the bin folder
- system documentation files

See https://github.com/apache/karaf/tree/master/examples/karaf-maven-example/karaf-maven-example-assembly for a custom distribution example.

5.8.1. Maven assembly

The recommended way to create a Karaf server assembly is to use the karaf-assembly packaging with the karaf-maven-plugin. This assembles a server from the maven dependencies in the project pom. After explanation of the configuration options we present an example.

The Karaf project effectively uses this packaging to assemble the official Karaf distributions, but due to maven limitations we have to simulate rather than use the karaf-assembly packaging.

This packaging creates tar.gz and zip archives containing the assembled server. They are identical except that zip archives don't unpack with appropriate unix file permissions for the scripts.

Maven dependencies

Maven dependencies in a karaf-assembly project can be feature repositories (classifier "features") or kar archives. Feature repositories are installed in the internal "system" Maven structured repository. Kar archives have their content unpacked on top of the server as well as contained feature repositories installed.

The Maven scope of a dependency determines whether its feature repository is listed in the features service configuration file `etc/org.apache.karaf.features.cfg` `featuresRepositories` property:

- compile (default): All the features in the repository (or for a kar repositories) will be installed into the `startup.properties`. The feature repo is not listed in the features service configuration file.
- runtime: feature installation is controlled by `<startupFeature>`, `<bootFeature>`, and `<installedFeature>` elements in the karaf-maven-plugin configuration. The feature repo uri is listed in the features service configuration file.

Plugin configuration

Control how features are installed using these elements referring to features from installed feature repositories:

- `<startupFeature>foo</startupFeature>` - This will result in the feature bundles being listed in startup.properties at the appropriate start level and the bundles being copied into the "system" internal repository. You can use feature_name or feature_name/feature_version formats.
- `<bootFeature>bar</bootFeature>` - This will result in the feature name added to boot-features in the features service configuration file and all the bundles in the feature copied into the "system" internal repository. You can use feature_name or feature_name/feature_version formats.
- `<installedFeature>baz</installedFeature>` - This will result in all the bundles in the feature being installed in the "system" internal repository. Therefore at runtime the feature may be installed without access to external repositories. You can use feature_name or feature_name/feature_version formats.

You can also define the libraries shipped in your custom distribution. For instance, it could be interesting if you want to extend your distribution with some JDBC drivers.

The plugin accepts the `<libraries/>` element where you can add `<library/>` containing the URL of the library. For instance:

```
<libraries>
    <library>mvn:org.postgresql/postgresql/9.3-1102-jdbc41;type:=endorsed</library>
</libraries>
```

Minimal Distribution Example

This is the minimal assembly pom changed to use the packaging and annotated

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

    <parent>
...
    </parent>

    <groupId>org.apache.karaf</groupId>
    <artifactId>apache-karaf-minimal</artifactId>
    <version>${project.version}</version>
    <packaging>karaf-assembly</packaging>
    <name>Apache Karaf :: Assemblies :: Minimal Distribution</name>

    <dependencies>
        <dependency>
        <!-- scope is compile so all features (there is only one) are installed into
startup.properties and the feature repo itself is not added in etc/
org.apache.karaf.features.cfg file -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>framework</artifactId>
            <version>${project.version}</version>
            <type>kar</type>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in etc/
org.apache.karaf.features.cfg file, and features will installed into the system directory
-->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>standard</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <!-- if you want to include resources in the distribution -->
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>false</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
            <resource>
```

```
                <directory>src/main/filtered-resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>

        <plugins>
            <!-- karaf-maven-plugin will call both assembly and archive goals -->
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <!-- no startupFeatures -->
                    <bootFeatures>
                        <feature>standard</feature>
                    </bootFeatures>
                    <!-- no installedFeatures -->
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Custom Distribution Example

It's possible to specify feature versions using the name/version format.

For instance, to pre-install Spring 4.0.7.RELEASE_1 feature in your custom distribution, you can use the following pom.xml:

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

    <modelVersion>4.0.0</modelVersion>

    <groupId>my.custom</groupId>
    <artifactId>my.distribution</artifactId>
    <version>1.0</version>
    <packaging>karaf-assembly</packaging>

    <dependencies>
        <dependency>
        <!-- scope is compile so all features (there is only one) are installed into
startup.properties and the feature repo itself is not added in etc/
org.apache.karaf.features.cfg file -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>framework</artifactId>
            <version>4.0.0</version>
            <type>kar</type>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in etc/
org.apache.karaf.features.cfg file, and features will installed into the system directory
if specify in the plugin configuration -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>standard</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
        <dependency>
        <!-- scope is runtime so the feature repo is listed in etc/
org.apache.karaf.features.cfg file, and features will installed into the system directory
if specify in the plugin configuration -->
            <groupId>org.apache.karaf.features</groupId>
            <artifactId>spring</artifactId>
            <classifier>features</classifier>
            <type>xml</type>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <!-- if you want to include resources in the distribution -->
        <resources>
            <resource>
                <directory>src/main/resources</directory>
```

```
                <filtering>false</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/filtered-resources</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*</include>
                </includes>
            </resource>
        </resources>

        <plugins>
            <plugin>
                <groupId>org.apache.karaf.tooling</groupId>
                <artifactId>karaf-maven-plugin</artifactId>
                <version>4.0.0</version>
                <extensions>true</extensions>
                <configuration>
                    <!-- no startupFeatures -->
                    <bootFeatures>
                      <feature>minimal</feature>
                    </bootFeatures>
                    <installedFeatures>
                        <feature>wrapper</feature>
                        <feature>spring/4.0.7.RELEASE_1</feature>
                    </installedFeatures>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

5.8.2. (deprecated old style) Maven assembly

Basically a Karaf custom distribution involves:

1. Uncompressing a standard Karaf distribution in a given directory.
2. Populating the system repo with your features.
3. Populating the lib directory with your branding or other system bundle jar files.
4. Overriding the configuration files in the etc folder.

These tasks could be performed using scripting, or more easily and portable, using Apache Maven and a set of Maven plugins.

For instance, the Maven POM could look like:

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

  <groupId>my.company</groupId>
  <artifactId>mycustom-karaf</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <name>My Unix Custom Karaf Distribution</name>

  <properties>
    <karaf.version>${project.version}</karaf.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.karaf</groupId>
      <artifactId>apache-karaf</artifactId>
      <version>${karaf.version}</version>
      <type>tar.gz</type>
    </dependency>
    <dependency>
      <groupId>org.apache.karaf</groupId>
      <artifactId>apache-karaf</artifactId>
      <version>${karaf.version}</version>
      <type>xml</type>
      <classifier>features</classifier>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <resource>
        <directory>${project.basedir}/src/main/filtered-resources</directory>
        <filtering>true</filtering>
        <includes>
          <include>**/*</include>
        </includes>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
        <executions>
          <execution>
            <id>filter</id>
            <phase>generate-resources</phase>
```

```
            <goals>
              <goal>resources</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>${karaf.version}</version>
        <executions>
          <execution>
           <id>add-features-to-repo</id>
           <phase>generate-resources</phase>
           <goals>
             <goal>features-add-to-repo</goal>
           </goals>
           <configuration>
              <descriptors>
                <descriptor>mvn:org.apache.karaf/apache-karaf/${karaf.version}/xml/
features</descriptor>
                <descriptor>file:${project.basedir}/target/classes/my-
features.xml</descriptor>
              </descriptors>
              <features>
                <feature>my-feature</feature>
              </features>
           </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy</id>
            <phase>generate-resources</phase>
            <goals>
              <goal>copy</goal>
            </goals>
            <configuration>
               <!-- Define here the artifacts which should be considered in the assembly
-->
               <!-- For instance, the branding jar -->
               <artifactItems>
                 <artifactItem>
                    <groupId>my.groupId</groupId>
                    <artifactId>my.branding.id</artifactId>
```

```
                <version>1.0</version>
                <outputDirectory>target/dependencies</outputDirectory>
                <destFileName>mybranding.jar</destFileName>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
        <execution>
          <!-- Uncompress the standard Karaf distribution -->
          <id>unpack</id>
          <phase>generate-resources</phase>
          <goals>
            <goal>unpack</goal>
          </goals>
          <configuration>
            <artifactItems>
              <artifactItem>
                <groupId>org.apache.karaf</groupId>
                <artifactId>apache-karaf</artifactId>
                <type>tar.gz</type>
                <outputDirectory>target/dependencies</outputDirectory>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <executions>
        <execution>
          <id>bin</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
          <configuration>
            <descriptors>
              <descriptor>src/main/descriptors/bin.xml</descriptor>
            </descriptors>
            <appendAssemblyId>false</appendAssemblyId>
            <tarLongFileMode>gnu</tarLongFileMode>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
</project>
```

The Maven POM will download the Karaf standard distribution and prepare resources to be processed by the Maven assembly plugin.

Your Maven project structure should look like:

- pom.xml: the previous POM file
- src/main/descriptors/bin.xml: the assembly Maven plugin descriptor (see below)
- src/main/filtered-resources: contains all resource files that have Maven property values to be filtered/replaced. Typically, this will include features descriptor and configuration files.
- src/main/distribution: contains all raw files which will be copied as-is into your custom distribution.

For instance, `src/main/filtered-resources` could contain:

- `my-features.xml` where Maven properties will be replaced
- `etc/org.apache.karaf.features.cfg` file containing your my-features descriptor:

```
#
# Comma separated list of features repositories to register by default
#
featuresRepositories=mvn:org.apache.karaf/apache-karaf/${karaf.version}/xml/
features,mvn:my.groupId/my-features/${project.version}/xml/features

#
# Comma separated list of features to install at startup
#
featuresBoot=config,ssh,management,my-feature
```

The `src/main/distribution` contains all your custom Karaf configuration files and script, as, for examples:

- etc/org.ops4j.pax.logging.cfg

```
# Root logger
log4j.rootLogger=INFO, out, osgi:VmLogAppender
log4j.throwableRenderer=org.apache.log4j.OsgiThrowableRenderer

# CONSOLE appender not used by default
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} | %-5.5p | %-16.16t |
%-32.32C %4L | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n

# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32C
%4L | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.home}/log/my-customer-distribution.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10

# Sift appender
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=my-custom
log4j.appender.sift.appender=org.apache.log4j.FileAppender
log4j.appender.sift.appender.layout=org.apache.log4j.PatternLayout
log4j.appender.sift.appender.layout.ConversionPattern=%d{ABSOLUTE} | %-5.5p | %-16.16t |
%-32.32c{1} | %-32.32C %4L | %m%n
log4j.appender.sift.appender.file=${karaf.log}/$\\{bundle.name\\}.log
log4j.appender.sift.appender.append=true
```

- etc/system.properties

```
#
# The properties defined in this file will be made available through system
# properties at the very beginning of the FAS boot process.
#

# Log level when the pax-logging service is not available
# This level will only be used while the pax-logging service bundle
# is not fully available.
# To change log levels, please refer to the org.ops4j.pax.logging.cfg file
# instead.
org.ops4j.pax.logging.DefaultServiceLog.level=ERROR

#
# Name of this custom instance.
#
karaf.name=my-custom

#
# Default repository where bundles will be loaded from before using
# other Maven repositories. For the full Maven configuration, see the
# org.ops4j.pax.url.mvn.cfg file.
#
karaf.default.repository=system

#
# Location of a shell script that will be run when starting a shell
# session. This script can be used to create aliases and define
# additional commands.
#
karaf.shell.init.script=${karaf.home}/etc/shell.init.script

#
# Set this empty property to avoid errors when validating xml documents.
#
xml.catalog.files=

#
# Suppress the bell in the console when hitting backspace to many times
# for example
#
jline.nobell=true

#
# Default port for the OSGi HTTP Service
#
org.osgi.service.http.port=8181

#
# Allow usage of ${custom.home} as an alias for ${karaf.home}
```

```
#
custom.home=${karaf.home}
```

- etc/users.properties

```
admin=admin,admin
```

- You can add a `etc/custom.properties`, it's a placeholder for any override you may need. For instance:

```
karaf.systemBundlesStartLevel=50
obr.repository.url=http://svn.apache.org/repos/asf/servicemix/smx4/obr-repo/repository.xml
org.osgi.framework.system.packages.extra = \
  org.apache.karaf.branding; \
  com.sun.org.apache.xalan.internal.xsltc.trax; \
  com.sun.org.apache.xerces.internal.dom; \
  com.sun.org.apache.xerces.internal.jaxp; \
  com.sun.org.apache.xerces.internal.xni; \
  com.sun.jndi.ldap
```

Now, we can "assemble" our custom distribution using the Maven assembly plugin. The Maven assembly plugin uses an assembly descriptor, configured in POM above to be `src/main/descriptors/bin.xml`:

```
<assembly>

    <id>bin</id>

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

    <fileSets>

        <!-- Expanded Karaf Standard Distribution -->
        <fileSet>
            <directory>target/dependencies/apache-karaf-${karaf.version}</directory>
            <outputDirectory>/</outputDirectory>
            <excludes>
                <exclude>bin/**</exclude>
                <exclude>etc/system.properties</exclude>
                <exclude>etc/users.properties</exclude>
                <exclude>etc/org.apache.karaf.features.cfg</exclude>
                <exclude>etc/org.ops4j.pax.logging.cfg</exclude>
                <exclude>LICENSE</exclude>
                <exclude>NOTICE</exclude>
                <exclude>README.md</exclude>
                <exclude>RELEASE-NOTES.md</exclude>
                <exclude>karaf-manual*.html</exclude>
                <exclude>karaf-manual*.pdf</exclude>
            </excludes>
        </fileSet>

        <!-- Copy over bin/* separately to get the correct file mode -->
        <fileSet>
            <directory>target/dependencies/apache-karaf-${karaf.version}</directory>
            <outputDirectory>/</outputDirectory>
            <includes>
                <include>bin/admin</include>
                <include>bin/karaf</include>
                <include>bin/start</include>
                <include>bin/stop</include>
            </includes>
            <fileMode>0755</fileMode>
        </fileSet>

        <!-- Copy over jar files -->
        <fileSet>
            <directory>target/dependencies</directory>
            <includes>
                <include>my-custom.jar</include>
            </includes>
            <outputDirectory>/lib/</outputDirectory>
```

```
        </fileSet>

        <fileSet>
            <directory>src/main/distribution</directory>
            <outputDirectory>/</outputDirectory>
            <fileMode>0644</fileMode>
        </fileSet>
        <fileSet>
            <directory>target/classes/etc</directory>
            <outputDirectory>/etc/</outputDirectory>
            <lineEnding>unix</lineEnding>
            <fileMode>0644</fileMode>
        </fileSet>

        <fileSet>
            <directory>target/features-repo</directory>
            <outputDirectory>/system</outputDirectory>
        </fileSet>

    </fileSets>

    <files>
        <file>
            <source>${basedir}/target/dependencies/apache-karaf-${karaf.version}/bin/
karaf</source>
            <outputDirectory>/bin/</outputDirectory>
            <destName>my-custom</destName>
            <fileMode>0755</fileMode>
            <lineEnding>unix</lineEnding>
        </file>
        <file>
            <source>${basedir}/target/classes/features.xml</source>
            <outputDirectory>/system/my.groupid/my-
features/${project.version}</outputDirectory>
            <destName>my-features-${project.version}-features.xml</destName>
            <fileMode>0644</fileMode>
            <lineEnding>unix</lineEnding>
        </file>
    </files>

</assembly>
```

To build your custom Karaf distribution, just run:

```
mvn install
```

You will find your Karaf custom distribution tar.gz in the target directory.

5.9. Creating Bundles

The examples provides different kind of bundles and services definition:

- https://github.com/apache/karaf/blob/master/examples/karaf-bundle-example/README.md
- https://github.com/apache/karaf/blob/master/examples/karaf-blueprint-example/README.md
- https://github.com/apache/karaf/blob/master/examples/karaf-scr-example/README.md

5.9.1. Add extended information to bundles

Karaf supports a OSGI-INF/bundle.info file in a bundle. This file is an extended description of the bundle. It supports ASCII character declarations (for adding color, formatting, etc) and some simple Wiki syntax.

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

```
h1. SYNOPSIS
    ${project.description}

h1. DESCRIPTION
    Long description of your bundle, including usage, etc.

h1. SEE ALSO
    [http://yourside\]
    [http://yourside/docs\]
```

You can display this extended information using:

```
root@karaf> bundles:info
```

5.9.2. Wiki Syntax

Karaf supports some simple wiki syntax in bundle info files:

```
h1., h2., ... : Headings
* : Enumerations
[ http://.... ] : links (there must be a space after the opening bracket)
**text** : bold text
```

5.10. Creating bundles for non OSGi third party dependencies

5.10.1. Dynamically wrapping jars

Karaf supports the wrap: protocol execution.

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

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

The wrap protocol creates a bundle dynamically using the bnd. Configurations can be added in the wrap URL:

- from the shell

```
root@karaf> bundles:install 'wrap:mvn:commons-lang/commons-lang/2.4$Bundle-
SymbolicName=commons-lang&Bundle-Version=2.4'
```

- from features.xml

```
<bundle>wrap:mvn:commons-lang/commons-lang/2.4$Bundle-SymbolicName=commons-
lang&amp;Bundle-Version=2.4</bundle>
```

Important notice : Add as child of your feature definition, the reference to wrap feature

```
<feature prerequisite="true">wrap</feature>
```

Additional information about meaning of `prerequisite` attribute can be found in Feature prerequisites description.

For instance :

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.4.0" name="app-2.0.0">
    <feature name="external-libs" version="2.0.0" description="External libs">
        <details>External dependencies</details>
        <feature prerequisite="true">wrap</feature>
        <bundle start-level="80">wrap:mvn:net.sf.ehcache/ehcache-core/2.6.11$Bundle-
SymbolicName=ehcache-core&amp;Bundle-Version=2.6.11</bundle>
    </feature>
</features>
```

5.10.2. Statically bundling jars

You can also create a wrap bundle for a third party dependency. This bundle is simply a Maven POM that shades an existing jar and package into a jar bundle.

For instance, to create an OSGi bundle that wraps Apache Commons Lang, simply define the following Maven POM:

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

    <modelVersion>4.0.0</modelVersion>

    <groupId>osgi.commons-lang</groupId>
    <artifactId>osgi.commons-lang</artifactId>
    <version>2.4</version>
    <packaging>bundle</packaging>
    <name>commons-lang OSGi Bundle</name>
    <description>This OSGi bundle simply wraps commons-lang-2.4.jar
artifact.</description>

    <dependencies>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.4</version>
            <optional>true</optional>
        </dependency>
    </dependencies>

    <build>
        <defaultGoal>install</defaultGoal>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>1.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <artifactSet>
                            <includes>
                                <include>commons-lang:commons-lang</include>
                            </includes>
                        </artifactSet>
                        <filters>
                            <filter>
                                <artifact>commons-lang:commons-lang</artifact>
                                <excludes>
                                    <exclude>**</exclude>
```

```
                                </excludes>
                            </filter>
                        </filters>

<promoteTransitiveDependencies>true</promoteTransitiveDependencies>
                        <createDependencyReducedPom>true</createDependencyReducedPom>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <version>2.1.0</version>
            <extensions>true</extensions>
            <configuration>
                <instructions>
                    <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                    <Export-Package></Export-Package>
                    <Import-Package></Import-Package>

<_versionpolicy>[$(version;==;$(@)),$(version;+;$(@)))</_versionpolicy>
                    <_removeheaders>Ignore-Package,Include-Resource,Private-Package,Embed-
Dependency</_removeheaders>
                </instructions>
                <unpackBundle>true</unpackBundle>
            </configuration>
        </plugin>
    </build>

</project>
```

The resulting OSGi bundle can now be deployed directly:

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

5.11. Blueprint

See https://github.com/apache/karaf/blob/master/examples/karaf-blueprint-example/README.md

5.12. SCR

See https://github.com/apache/karaf/tree/master/examples/karaf-scr-example

5.13. Archetypes

Karaf provides archetypes to easily create commands, manage features or repository and create a kar archive. This section describes each of them and explain How to use it.

5.13.1. Create a command (karaf-command-archetype)

This archetype creates a Maven skeleton project that you will use to develop new Karaf commands.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.karaf.archetypes \
  -DarchetypeArtifactId=karaf-command-archetype \
  -DarchetypeVersion=4.0.0 \
  -DgroupId=com.mycompany \
  -DartifactId=com.mycompany.command \
  -Dversion=1.0-SNAPSHOT \
  -Dpackage=com.mycompany.package
```

Additional parameters

During the maven creation process, additional questions will be asked on the console :

- Define value for property *command*: the name of the command (list, add-item, ...)
- Define value for property *description*: provide a description of the command that you want to create. This description will be displayed in the Karaf console when the parameter --help is used
- Define value for property *scope*: this value represents the family name to which the command belongs (features, osgi, admin, jaas, ...)

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-command-archetype:4.0.0] found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.command
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Define value for property 'command': : list
Define value for property 'description': : List sample command
Define value for property 'scope': : my
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.command
version: 1.0-SNAPSHOT
package: com.mycompany.package
command: list
description: List sample command
scope: my
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: karaf-command-
archetype:4.0.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.command
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: scope, Value: my
[INFO] Parameter: description, Value: List sample command
[INFO] Parameter: command, Value: list
[INFO] Parameter: artifactId, Value: com.mycompany.command
[WARNING] Don't override file /com.mycompany.command/pom.xml
[INFO] project created from Archetype in dir: /com.mycompany.command
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 27.204s
[INFO] Finished at: Mon Dec 19 09:38:49 CET 2011
[INFO] Final Memory: 7M/111M
[INFO] ------------------------------------------------------------------------
```

Next, you can import your project in Eclipse/IntelliJ and developp the Karaf command.

5.13.2. Create an OSGi bundle (karaf-bundle-archetype)

This archetype creates a Maven skeleton to create an OSGi bundle, including a bundle Activator (a special callback class for bundle start/stop).

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-bundle-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.bundle \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.package
```

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-bundle-archetype:4.0.0] found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.bundle
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.bundle
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: karaf-bundle-
archetype:4.0.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.bundle
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.bundle
[INFO] project created from Archetype in dir: /com.mycompany.bundle
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.895s
[INFO] Finished at: Mon Dec 19 11:41:44 CET 2011
[INFO] Final Memory: 8M/111M
[INFO] ------------------------------------------------------------------------
```

5.13.3. Create an OSGi blueprint bundle (karaf-blueprint-archetype)

This archetype creates a Maven skeleton project to create an OSGi blueprint bundle, including a sample bean exposed as an OSGi service in the blueprint XML descriptor.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-blueprint-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.blueprint \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.blueprint
```

Result of Maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-blueprint-archetype:4.0.0] found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.blueprint
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.blueprint
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: karaf-blueprint-
archetype:4.0.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.blueprint
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.blueprint
[INFO] project created from Archetype in dir: /com.mycompany.blueprint
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1:06:36.741s
[INFO] Finished at: Mon Dec 19 13:04:43 CET 2011
[INFO] Final Memory: 7M/111M
[INFO] ------------------------------------------------------------------------
```

5.13.4. Create a features XML (karaf-feature-archetype)

This archetype creates a Maven skeleton project which create a features XML file, using the dependencies that you define in the POM.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-feature-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=my.company \
    -DartifactId=my.company.feature \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=my.company.package
```

Result of maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-feature-archetype:4.0.0] found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.feature
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.feature
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: karaf-feature-
archetype:4.0.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.feature
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.feature
[INFO] project created from Archetype in dir: /com.mycompany.feature
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.262s
[INFO] Finished at: Mon Dec 19 13:20:00 CET 2011
[INFO] Final Memory: 7M/111M
[INFO] ------------------------------------------------------------------------
```

5.13.5. Create a KAR file (karaf-kar-archetype)

This archetype creates a Maven skeleton project including a features XML sample, used to generate a KAR file.

Command line

Using the command line, we can create our project:

```
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.karaf.archetypes \
    -DarchetypeArtifactId=karaf-kar-archetype \
    -DarchetypeVersion=4.0.0 \
    -DgroupId=com.mycompany \
    -DartifactId=com.mycompany.kar \
    -Dversion=1.0-SNAPSHOT \
    -Dpackage=com.mycompany.package
```

Result of maven command execution

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[org.apache.karaf.archetypes:karaf-kar-archetype:4.0.0] found in catalog local
[INFO] Using property: groupId = com.mycompany
[INFO] Using property: artifactId = com.mycompany.kar
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = com.mycompany.package
Confirm properties configuration:
groupId: com.mycompany
artifactId: com.mycompany.kar
version: 1.0-SNAPSHOT
package: com.mycompany.package
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Archetype: karaf-kar-
archetype:4.0.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.kar
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: packageInPathFormat, Value: com/mycompany/package
[INFO] Parameter: package, Value: com.mycompany.package
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: artifactId, Value: com.mycompany.kar
[INFO] project created from Archetype in dir: /com.mycompany.kar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.465s
[INFO] Finished at: Mon Dec 19 13:30:15 CET 2011
[INFO] Final Memory: 8M/157M
[INFO] ------------------------------------------------------------------------
```

5.14. Security framework

Karaf supports JAAS (http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html) 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.

5.14.1. Overview

This feature allows runtime deployment of JAAS based configuration for use in various parts of the application. This includes the remote console login, which uses the `karaf` realm, but which is configured with a dummy login module by default. These realms can also be used by the NMR, JBI components or the JMX server to authenticate users logging in or sending messages into the bus.

In addition to JAAS realms, you can also deploy keystores and truststores to secure the remote shell console, setting up HTTPS connectors or using certificates for WS-Security.

A very simple XML schema for spring has been defined, allowing the deployment of a new realm or a new keystore very easily.

5.14.2. Schema and Deployer

To override or deploy a new realm, you can use the following XSD which is supported by a Blueprint namespace handler and can thus be defined in a Blueprint xml configuration file. You can directly create a XML file in the Apache Karaf `deploy` folder. To enable this JAAS config deployer, you have first to install the `jaas-deployer` feature:

```
karaf@root()> feature:install jaas-deployer
```

You can find the schema at the following http://karaf.apache.org/xmlns/jaas/v1.1.0

Here are two examples using this schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

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

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

</blueprint>
```

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

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

Each realm can contain one or more module definitions. Each module identifies a LoginModule and the `className` attribute must be set to the class name of the login module to use. Note that this login module must be available from the bundle classloader, so either it has to be defined in the bundle itself, or the needed package needs to be correctly imported. The `flags` attribute can take one of four values. The content of the `module` element is parsed as a properties file and will be used to further configure the login module.

Deploying such a code will lead to a JaasRealm object in the OSGi registry, which will then be used when using the JAAS login module.

Configuration override and use of the `rank` attribute

The `rank` attribute on the `config` element is tied to the ranking of the underlying OSGi service. When the JAAS framework performs an authentication, it will use the realm name to find a matching JAAS configuration. If multiple configurations are used, the one with the highest `rank` attribute will be used. So if you want to override the default security configuration in Karaf (which is used by the ssh shell, web console and JMX layer), you need to deploy a JAAS configuration with the name `name="karaf"` and `rank="1"`.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

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

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

</blueprint>
```

5.14.3. 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 defined LoginModule. If you plan to integrate this feature into another OSGi runtime, this class must be made available from the system classloader and the related package be part of the boot delegation classpath (or be deployed as a fragment attached to the system bundle).

The xml schema defined above allows the use of a simple xml (leveraging spring xml extensibility) to configure and register a JAAS configuration for a given realm. This configuration will be made available into the OSGi registry as a JaasRealm and the OSGi specific Configuration will look for such services. Then the proxy login module will be able to use the information provided by the realm to actually load the class from the bundle containing the real login module.

Karaf itself provides a set of login modules ready to use, depending of the authentication backend that you need.

In addition to the login modules, Karaf also supports backend engines. The backend engine is coupled to a login module and allows you to manipulate users and roles directly from Karaf (adding a new user, delete an existing user, etc). The backend engine is constructed by a backend engine factory, registered as an OSGi service. Some login modules (for security reasons for instance) don't provide backend engine.

5.14.4. Available realm and login modules

Karaf comes with a default realm named "karaf" using login modules.

Karaf also provides a set of login modules and backend engines to handle authentication needs for your environment.

PropertiesLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule	org.apache.karaf.jaas.modules.pro

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

Name	Description
`users`	location of the properties file

This file uses the properties file format (http://download.oracle.com/javase/6/docs/api/java/util/Properties.html#load(java.io.Reader)). The format of the properties is as follows, with each line defining a user, its password and associated roles:

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

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

The PropertiesLoginModule provides a backend engine allowing:

- add a new user
- delete an existing user
- list the users, groups, and roles
- add a new role to an user
- delete a role from an user
- add an user into a group
- remove an user from a group
- add a role to a group
- delete a role from a group

To enable the backend engine, you have to register the corresponding OSGi service. For instance, the following blueprint shows how to register the PropertiesLoginModule and the corresponding backend engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

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

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

</blueprint>
```

OsgiConfigLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule	N/A

The OsgiConfigLoginModule uses the OSGi ConfigurationAdmin service to provide the users, passwords and roles.

Name	Description
`pid`	the PID of the configuration containing user definitions

The format of the configuration is the same than for the `PropertiesLoginModule` with properties prefixed with `user.`.

For instance, in the Karaf etc folder, we create a file `org.apache.karaf.authentication.cfg` containing:

```
user.karaf=karaf,admin
user.user=password,role
```

The following blueprint shows how to use this configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0">

    <jaas:config name="karaf" rank="-1">
        <jaas:module className="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule"
                     flags="required">
            pid = org.apache.karaf.authentication
        </jaas:module>
    </jaas:config>

</blueprint>
```

NOTE | The OsgiConfigLoginModule doesn't provide a backend engine.

JDBCLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule	org.apache.karaf.jaas.modules.jdbc.JDBCBack

The JDBCLoginModule uses a database to load the users, passwords and roles from a provided data source (normal or XA). The data source and the queries for password and role retrieval are configurable using the following parameters.

Name	Description
`datasource`	The datasource as on OSGi ldap filter or as JNDI 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

To use an OSGi ldap filter, the prefix osgi: needs to be provided, as shown below:

```
<jaas:config name="karaf">
    <jaas:module className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                 flags="required">
        datasource = osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
        query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
        query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
    </jaas:module>
</jaas:config>
```

To use an JNDI name, the prefix jndi: needs to be provided. The example below assumes the use of Aries jndi to expose services via JNDI.

```
<jaas:config name="karaf">
    <jaas:module className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                 flags="required">
        datasource = jndi:aries:services/
javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
        query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
        query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
    </jaas:module>
</jaas:config>
```

The JDBCLoginModule provides a backend engine allowing:

- add a new user
- delete an user
- list users, roles
- add a new role to an user
- remove a role from an user

> **NOTE** The groups are not fully supported by the JDBCBackingEngine.

The following blueprint shows how to define the JDBCLoginModule with the corresponding backend engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0">

    <jaas:config name="karaf">
        <jaas:module className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
                flags="required">
            datasource = jndi:aries:services/
javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
            query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
            query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
            insert.user = INSERT INTO USERS(USERNAME,PASSWORD) VALUES(?,?)
            insert.role = INSERT INTO ROLES(ROLE,USERNAME) VALUES(?,?)
            delete.user = DELETE FROM USERS WHERE USERNAME=?
        </jaas:module>
    </jaas:config>

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

</blueprint>
```

LDAPLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.ldap.LDAPLoginModule	N/A

The LDAPLoginModule uses LDAP to load the users and roles and bind the users on the LDAP to check passwords.

The LDAPLoginModule supports the following parameters:

Name	Description
`connection.url`	The LDAP connection URL, e.g. ldap://hostname
`connection.username`	Admin username to connect to the LDAP. This parameter is optional, if it's not provided, the LDAP connection will be anonymous.

Name	Description
`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", th 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", th role lookup will be performed only at the first level (ONELEVEL).
`role.mapping`	Define a mapping between roles defined in the LDAP directory fo the user, and corresponding roles in Karaf. The format is ldapRole1=karafRole1,karafRole2;ldapRole2=karafRole3,karafRol
`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

Name	Description
`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.
`ignorePartialResultException`	Workaround for Active Directory servers not handling referrals correctly. When the `context.java.naming.referral` parameter is `false` (as it is by default), this will prevent `PartialResultException`s from being thrown during enumeration of search results.

A example of LDAPLoginModule usage follows:

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

If you wish to use an SSL connection, the following configuration can be used as an example:

```
<ext:property-placeholder />

<jaas:config name="karaf" rank="1">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
        connection.url = ldaps://localhost:10636
        user.base.dn = ou=users,ou=system
        user.filter = (uid=%u)
        user.search.subtree = true
        role.base.dn = ou=groups,ou=system
        role.filter = (uniqueMember=uid=%u)
        role.name.attribute = cn
        role.search.subtree = true
        authentication = simple
        ssl.protocol=SSL
        ssl.truststore=ks
        ssl.algorithm=PKIX
    </jaas:module>
</jaas:config>

<jaas:keystore name="ks"
               path="file:///${karaf.home}/etc/trusted.ks"
               keystorePassword="secret" />
```

The LDAPLoginModule supports the following patterns that you can use in the filter (user and role filters):

- `%u` is replaced by the user
- `%dn` is replaced by the user DN
- `%fqdn` is replaced by the user full qualified DN (`userDNNamespace`).

For instance, the following configuration will work properly with ActiveDirectory (adding the ActiveDirectory to the default `karaf` realm):

```
<jaas:config name="karaf" rank="2">
  <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connection.username=admin
    connection.password=xxxxxxx
    connection.protocol=
    connection.url=ldap://activedirectory_host:389
    user.base.dn=ou=Users,ou=there,DC=local
    user.filter=(sAMAccountName=%u)
    user.search.subtree=true
    role.base.dn=ou=Groups,ou=there,DC=local
    role.name.attribute=cn
    role.filter=(member=%fqdn)
    role.search.subtree=true
    authentication=simple
  </jaas:module>
</jaas:config>
```

NOTE

The LDAPLoginModule doesn't provide backend engine. It means that the administration of the users and roles should be performed directly on the LDAP backend.

KerberosLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.krb5.Krb5LoginModule	

The Kerberos login module uses the Oracle JVM Krb5 internal login module.

Here is a simple configuration :

```
<jaas:config name="krb5" rank="1">
  <jaas:module className="org.apache.karaf.jaas.modules.krb5.Krb5LoginModule">
    refreshKrb5Config = true
    password-stacking = storePass
    doNotPrompt = false
    useTicketCache = true
  </jaas:module>
</jaas:config>
```

You must specify a krb5 configuration file through the "java.security.krb5.conf" system property. Here is a simple example of a krb5 configuration file :

```
[libdefaults]
 default_realm = EXAMPLE.COM
 dns_lookup_realm = false
 dns_lookup_kdc = false
 ticket_lifetime = 24h
 renew_lifetime = 365d
 forwardable = true

[realms]

 EXAMPLE.COM = {
  kdc = kdc.example.com
  admin_server = kdc.example.com
  default_domain = example.com
 }

[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
```

GSSAPILdapLoginModule

LoginModule	BackendEngineFactory
org.apache.karaf.jaas.modules.ldap.GSSAPILdapLoginModule	

The GSSAPI module uses the GSSAPI mechanism to handle authentication to a LDAP server. Typical use is using this and a Kerberos Login Module to connect to an ActiveDirectory Server, or any other LDAP server that needs a Kerberos tickets for authentication.

Here is a simple configuration, that use as Kerberos login module as authentication backend :

```
<jaas:config name="ldap" rank="1">
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.GSSAPILdapLoginModule"flags="required">
    gssapiRealm=krb5
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connection.url=ldap://activedirectory_host:389
    user.base.dn=ou=Users,ou=there,DC=local
    user.filter=(sAMAccountName=%u)
    user.search.subtree=true
    role.base.dn=ou=Groups,ou=there,DC=local
    role.name.attribute=cn
    role.filter=(member=%fqdn)
    role.search.subtree=true
  </jaas:module>
</jaas:config>
<jaas:config name="krb5" rank="1">
  <jaas:module className="org.apache.karaf.jaas.modules.krb5.Krb5LoginModule">
    refreshKrb5Config = true
    password-stacking = storePass
    doNotPrompt = false
    useTicketCache = true
  </jaas:module>
</jaas:config>
```

Note the *gssapiRealm* property of the LDAP login module that match the name of the Kerberos Configuration.

SyncopeLoginModule

LoginModule	**BackendEngineFactory**
org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule	org.apache.karaf.jaas.modules.syncop

The Syncope login module uses the Syncope REST API to authenticate users and retrieve the roles.

The Syncope login module just requires one parameter:

Name	**Description**
`address`	Location of the Syncope REST API
`version`	Syncope backend version (could by "1.x" or "2.x"

Name	Description
`admin.user`	Admin username to administrate Syncope (only required by the backend engine)
`admin.password`	Admin password to administrate Syncope (only required by the backend engine)

The following snippet shows how to use Syncope with the karaf realm:

```
<jaas:config name="karaf" rank="2">
  <jaas:module className="org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule"
flags="required">
    address=http://localhost:9080/syncope/cxf
    version=1.x
    admin.user=admin
    admin.password=password
  </jaas:module>
</jaas:config>
```

SyncopeLoginModule comes with a backend engine allowing to manipulate users and roles. You have to register the SyncopeBackendEngineFactory service.

For security reason, the SyncopeLoginModule backend engine allows only to list users and roles. You can't create or delete users and roles directly from Karaf. To do it, you have to use the Syncope web console.

For instance, the following blueprint descriptor enables the SyncopeLoginModule and the backend engine factory:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0">

    <jaas:config name="karaf" rank="2">
        <jaas:module className="org.apache.karaf.jaas.modules.syncope.SyncopeLoginModule"
                     flags="required">
           address=http://localhost:9080/syncope/cxf
           version=1.x
           admin.user=admin
           admin.password=password
        </jaas:module>
    </jaas:config>

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

</blueprint>
```

5.14.5. 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. Jasypt section)
`encryption.enabled`	Boolean used to turn on encryption
`encryption.prefix`	Prefix for encrypted passwords
`encryption.suffix`	Suffix for encrypted passwords

Name	Description
`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`) (see Role discovery policies section)
`role.discriminator`	A discriminator value to be used by the role policy

A simple example follows:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                  flags="required">
        users = $[karaf.base]/etc/users.properties
        encryption.enabled = true
        encryption.algorithm = MD5
        encryption.encoding = hexadecimal
    </jaas:module>
</jaas:config>
```

Prefix and suffix

The login modules have the ability to support both encrypted and plain passwords at the same time. In some cases, some login modules may be able to encrypt the passwords on the fly and save them back in an encrypted form.

Jasypt

Karaf default installation comes with a simple encryption service which usually fullfill simple needs. However, in some cases, you may want to install the Jasypt (http://www.jasypt.org/) library which provides stronger encryption algorithms and more control over them.

To install the Jasypt library, the easiest way is to install the available feature:

```
karaf@root> features:install jasypt-encryption
```

It will download and install the required bundles and also register an `EncryptionService` for Jasypt in the OSGi registry.

When configuring a login module to use Jasypt, you need to specify the `encryption.name` property and set it to a value of `jasypt` to make sure the Jasypt encryption service will be used.

In addition to the standard properties above, the Jasypt service provides the following parameters:

Name	Description
`providerName`	Name of the `java.security.Provider` name to use for obtaining the digest algorithm
`providerClassName`	Class name for the security provider to be used for obtaining the digest algorithm
`iterations`	Number of times the hash function will be applied recursively
`saltSizeBytes`	Size of the salt to be used to compute the digest
`saltGeneratorClassName`	Class name of the salt generator

A typical realm definition using Jasypt encryption service would look like:

```
<jaas:config name="karaf">
    <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
                 flags="required">
        users = $[karaf.base]/etc/users.properties
        encryption.enabled = true
        encryption.name = jasypt
        encryption.algorithm = SHA-256
        encryption.encoding = base64
        encryption.iterations = 100000
        encryption.saltSizeBytes = 16
    </jaas:module>
</jaas:config>
```

Using encrypted property placeholders

When using blueprint framework for OSGi for configuring devices that requires passwords like JDBC datasources, it is undesirable to use plain text passwords in configuration files. To avoid this problem it is good to store database passwords in encrypted format and use encrypted property placeholders when ever possible.

Encrypted properties can be stored in plain properties files. The encrypted content is wrapped by an ENC() function.

```
#db.cfg / db.properties
db.url=localhost:9999
db.username=admin
db.password=ENC(zRM7Pb/NiKyCalroBz8CKw==)
```

The encrypted property placeholders can be used either by defining Apache Aries ConfigAdmin `property-placeholder` or by directly using the Apache Karaf `property-placeholder`. It has one child element `encryptor` that contains the actual Jasypt configuration. For detailed information on how to configure the different Jasypt encryptors, see the Jasypt documentation (http://www.jasypt.org/general-usage.html).

A typical definition using Jasypt encryption would look like:

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

  <!-- Configuration via ConfigAdmin property-placeholder -->
  <!-- the etc/*.cfg can contain encrypted values with ENC() function -->
  <cm:property-placeholder persistent-id="db" update-strategy="reload">
    <cm:default-properties>
      <cm:property name="encoded" value="ENC(${foo})"/>
    </cm:default-properties>
  </cm:property-placeholder>

  <!-- Configuration via properties file -->
  <!-- Instead of ConfigAdmin, we can load "regular" properties file from a location -->
  <!-- Again, the db.properties file can contain encrypted values with ENC() function -->
  <ext:property-placeholder>
    <ext:location>file:etc/db.properties</ext:location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES"/>
          <property name="passwordEnvName" value="ENCRYPTION_PASSWORD"/>
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

  <!-- ... -->

</blueprint>
```

Don’t forget to install the jasypt feature to add the support of the enc namespace:

```
karaf@root()> feature:install jasypt-encryption
```

5.14.6. 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 on 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

When the prefixed role policy is used the login module applies a configurable prefix *(property role.discriminator)* to the role, so that the application can identify the role's principals by its prefix. Example:

```
<jaas:config name="karaf">
    <jaas:module
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())));
       }
}
```

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());
        }
    }
}
```

5.14.7. Default role policies

The previous section describes how to leverage role policies. However, Karaf provides a default role policy, based on the following class names:

- org.apache.karaf.jaas.modules.UserPrincipal
- org.apache.karaf.jaas.modules.RolePrincipal
- org.apache.karaf.jaas.modules.GroupPrincipal

It allows you to directly handling the role class:

```
String rolePrincipalClass = "org.apache.karaf.jaas.modules.RolePrincipal";

for (Principal p : subject.getPrincipals()) {
    if (p.getClass().getName().equals(rolePrincipalClass)) {
        roles.add(p.getName());
    }
}
```

5.15. Troubleshooting, Debugging, Profiling, and Monitoring

5.15.1. 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:

1. Use `log:set` to dynamically change the global log level
2. Execute the problematic operation
3. Use `log:display` (or `log:display-exception` to display the log

Worst Case Scenario

If you end up with Karaf in a really bad state (i.e. you can not boot it anymore) or you just want to revert to a clean state quickly, you can safely remove the `data` directory just in the installation directory. This folder contains transient data and will be recreated if removed when you relaunch Karaf. You may also want to remove the files in the `deploy` folder to avoid them being automatically installed when Karaf is started the first time.

5.15.2. Debugging

Usually, the easiest way to debug Karaf or any application deployed onto it is to use remote debugging. Remote debugging can be easily activated by using the `debug` parameter on the command line.

```
> bin/karaf debug
```

or on Windows

```
> bin\karaf.bat debug
```

Another option is to set the `KARAF_DEBUG` environment variable to `TRUE`.

This can be done using the following command on Unix systems:

```
export KARAF_DEBUG=true
```

On Windows, use the following command

```
set KARAF_DEBUG=true
```

Then, you can launch Karaf using the usual way:

```
bin/karaf
```

or

```
bin\karaf.bat
```

Last, inside your IDE, connect to the remote application (the default port to connect to is 5005).

This option works fine when it is needed to debug a project deployed on top of Apache Karaf. Nevertheless, you will be blocked if you would like to debug the Karaf server itself.

In this case, you can use `debugs` option (s as syspend). 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.

```
bin/karaf debugs
```

and on Windows,

```
bin\karaf.bat debugs
```

Debugging Environment Variables

Variable	Description	Default
KARAF_DEBUG	Set to TRUE to enable debugging.	
EXTRA_JAVA_OPTS	Java options append to JAVA_OPTS	
JAVA_DEBUG_OPTS	Java options to enable debuging.	Calculated based on the OS
JAVA_DEBUG_PORT	Port used by the debugger	5005

5.15.3. Profiling

jVisualVM

You have to edit the `etc/config.properties` configuration file to add the jVisualVM package:

```
org.osgi.framework.bootdelegation=...,org.netbeans.lib.profiler.server
```

Run Karaf from the console, and you should now be able to connect using jVisualVM.

YourKit

You need a few steps to be able to profile Karaf using YourKit.

The first one is to edit the `etc/config.properties` configuration file and add the following property:

```
org.osgi.framework.bootdelegation=...,com.yourkit.*
```

Then, set the `JAVA_OPTS` environment variable:

```
export JAVA_OPTS='-Xmx512M -agentlib:yjpagent'
```

or, on Windows

```
set JAVA_OPTS='-Xmx512M -agentlib:yjpagent'
```

Run Karaf from the console, and you should now be able to connect using YourKit standalone or from your favorite IDE.

5.15.4. Monitoring

Karaf uses JMX for monitoring and management of all Karaf components.

The JMX connection could be:

- local using the process id

jconsole connect

- 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:

jconsole overview

- A thread overview:

jconsole threads

- A memory heap consumption, including "Perform GC" button:

jconsole memory

- A complete JVM summary, with all number of threads, etc:

jconsole summary

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:

jconsole admin

You can also manage Karaf features MBean to list, install, and uninstall Karaf features:

jconsole features

5.16. Configuration

See https://github.com/apache/karaf/blob/master/examples/karaf-config-example/README.md

5.17. Deployer

See https://github.com/apache/karaf/blob/master/examples/karaf-deployer-example/README.md

5.18. Dump extender

See https://github.com/apache/karaf/blob/master/examples/karaf-dump-example/README.md

5.19. JDBC & JPA

See https://github.com/apache/karaf/blob/master/examples/karaf-jdbc-example/README.md

See https://github.com/apache/karaf/blob/master/examples/karaf-jpa-example/README.md

5.20. JMS

See https://github.com/apache/karaf/blob/master/examples/karaf-jms-example/README.md

5.21. Custom log appender

See https://github.com/apache/karaf/blob/master/examples/karaf-log-appender-example/README.md

5.22. Custom JMX MBean

See https://github.com/apache/karaf/blob/master/examples/karaf-mbean-example/README.md

5.23. Working with profiles

See https://github.com/apache/karaf/blob/master/examples/karaf-profile-example/README.md

5.24. Security & JAAS

See https://github.com/apache/karaf/blob/master/examples/karaf-jaas-example/README.md

5.25. Servlet

See https://github.com/apache/karaf/blob/master/examples/karaf-servlet-example/README.md

5.26. WAR

See https://github.com/apache/karaf/blob/master/examples/karaf-war-example/README.md

5.27. HTTP Resources

See https://github.com/apache/karaf/blob/master/examples/karaf-http-resource-example/README.md

5.28. REST service

See https://github.com/apache/karaf/blob/master/examples/karaf-rest-example/README.md

5.29. SOAP service

See https://github.com/apache/karaf/blob/master/examples/karaf-soap-example/README.md

5.30. Websocket

See https://github.com/apache/karaf/blob/master/examples/karaf-websocket-example/README.md

5.31. Scheduler

See https://github.com/apache/karaf/blob/master/examples/karaf-scheduler-example/README.md

5.32. Quick example with Apache Camel

See https://github.com/apache/karaf/blob/master/examples/karaf-camel-example/README.md

5.33. Custom URL handler

See https://github.com/apache/karaf/blob/master/examples/karaf-url-namespace-handler-example/README.md

5.34. Integration Tests

See https://github.com/apache/karaf/blob/master/examples/karaf-itest-example/README.md

5.35. Distributions & Docker

See https://github.com/apache/karaf/blob/master/examples/karaf-docker-example/README.md

5.36. Github Contributions

Some people prefer to make contributions to karaf source via github. If you are one of them, this is for you!

5.36.1. Introduction

Apache Karaf is available as a periodically replicated mirror on: https://github.com/apache/karaf

5.36.2. Suggested Workflow

1. Make a fork of karaf repo github mirror
2. Do all your new work on your own karaf fork
3. When ready, file a Jira issue https://issues.apache.org/jira/browse/KARAF, attach the link to your github pull request, and ask for a review
4. One of karaf committers will discuss your pull request on github; and at some point your pull request will be accepted
5. When your pull request is accepted, squash it into a single commit and attach single patch file to the original jira, with ASF grant check box selected
6. Now pray to your favorite ASF committer to really accept the patch :-)
7. When your patch is committed to the svn, and you can verify it in the latest karaf snapshot, close your pull request on github

5.36.3. License Reminder

In order for your contributions to be accepted:

- All files must contain ASL license grant header
- You must select ASF grant check box when attaching patch to the jira

5.36.4. How to Generate a One-File-Patch Via Throw-Away Branch

Here is one way to generate squash of your commits:

http://stackoverflow.com/questions/616556/how-do-you-squash-commits-into-one-patch-with-git-format-patch

```
#
# 'archon' referers to karaf mirror
# 'origin' referers to your own fork
#

# attach karaf mirror as remote, if not done yet
git remote add archon https://github.com/apache/karaf

# fetch latest karaf mirror
git fetch archon

# ensure you are on your fork trunk
git checkout origin/trunk

# kill previous patch delivery, if you had one
git branch -D delivery

# make new delivery throw-away branch, based on latest karaf mirror
git branch delivery archon/trunk

# use it
git checkout delivery

# squash all your local development into a single commit
git merge --squash trunk

# commit it to the delivery branch
git commit -m "delivery"

# generate a patch file against the mirror
git format-patch archon/trunk
```

The root of your Karaf source now contains a file named "0001-delivery.patch.txt" (please attach the .txt ending;this will allow commiters to open your patch directly in the browser and give it a short look there) which you should attach to your karaf jira.

Last updated 2020-11-04 17:04:18 CET