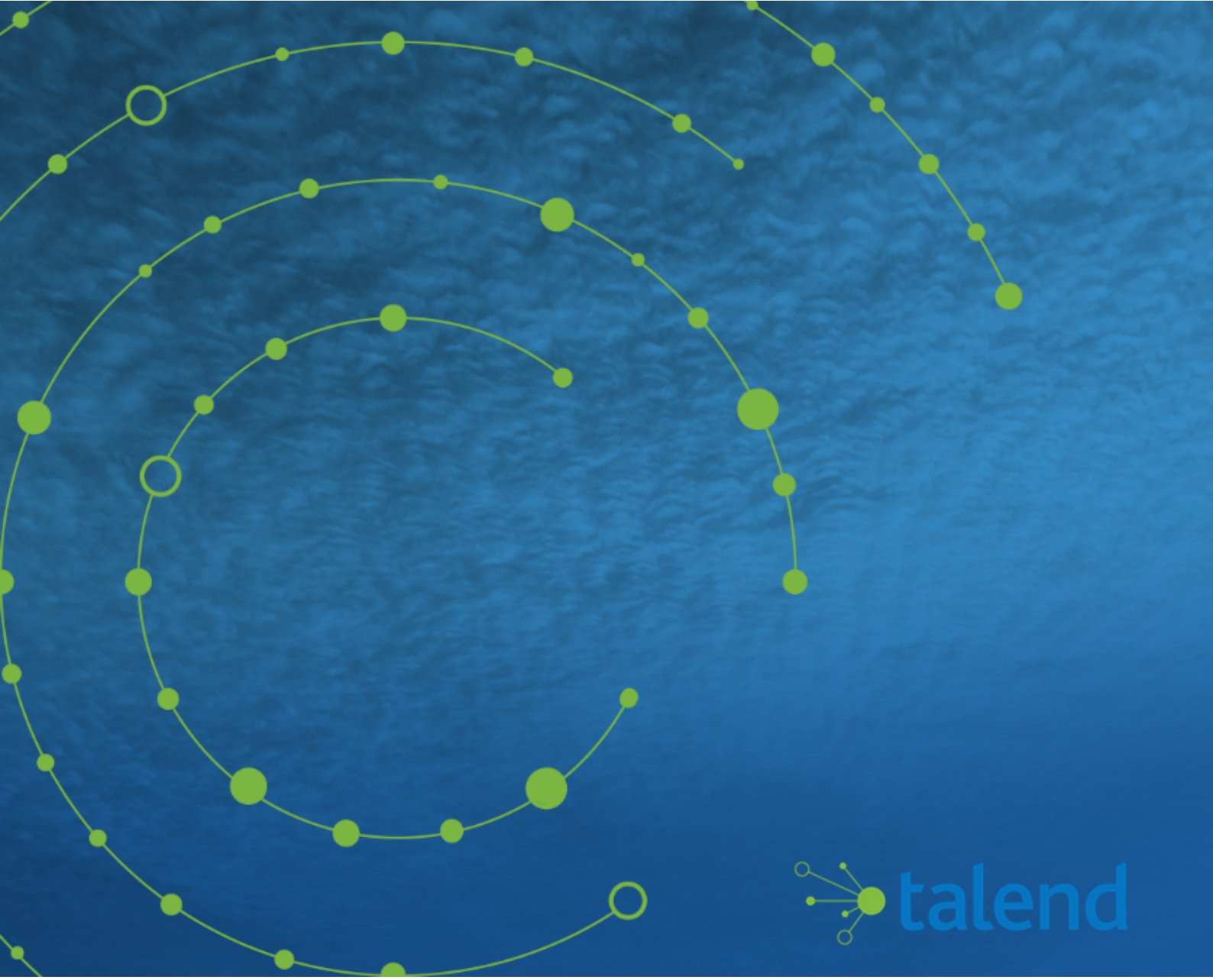


Talend ESB Container Administration Guide

7.3.1

Last updated: 2020-02-19

Contents

Copyright

Adapted for 7.3.1. Supersedes previous releases.

Copyright © 2020 Talend. All rights reserved.

The content of this document is correct at the time of publication.

However, more recent updates may be available in the online version that can be found on Talend Help Center.

Notices

Talend is a trademark of Talend, Inc.

All brands, product names, company names, trademarks and service marks are the properties of their respective owners.

End User License Agreement

The software described in this documentation is provided under **Talend** 's End User Software and Subscription Agreement ("Agreement") for commercial products. By using the software, you are considered to have fully understood and unconditionally accepted all the terms and conditions of the Agreement.

To read the Agreement now, visit http://www.talend.com/legal-terms/us-eula?utm_medium=help&utm_source=help_content

Introduction

Talend ESB provides an Apache Karaf-based ESB container preconfigured to support Apache Camel routing and Apache CXF-based services (both REST and SOAP-based). This container administration manual is intended to provide information on Karaf tooling commands and general administration.

Structure of this manual

The following chapters are in this manual:

- Release directory structure on page 7 describes the directory layout of a Talend ESB installation.
- Starting/Stopping Talend Runtime and its ESB Infrastructure Components on page 8 describes the various options that are available to start and stop Talend ESB.
- Using the console commands on page 10 gives a list of the console commands available.
- Deploying Multiple Karaf Containers on page 24 describes how to adapt the configuration to deploy multiple containers at the same machine.
- Remote Console on page 26 describes how to access Talend ESB using a remote console.
- Apache CXF and Camel, commands and tuning on page 29 describes commands provided by CXF and Camel.
- Security on page 33 describes how to configure security with Talend ESB.
- HTTP Configuration on page 45 describes how to configure HTTP in Talend Runtime.
- Logging System on page 56 describes configuring and using the logging system.
- Deployer on page 58 describes deploying bundles or re-defined groups of bundles (called features).
- Servlet Context on page 61 describes the servlet context configuration.
- Provisioning on page 62 describes how the provisioning system uses xml repositories that define sets of features (bundles). Thus when you install a feature and it will install all of its bundles defined under it.
- Web Applications on page 68 describes deploying WAR-based web applications into Talend Runtime.
- Monitoring and Administration using JMX on page 70 lists the MBeans available to monitor and administrate Karaf using any JMX client.
- Installing the Talend Runtime Container as a service on page 71 describes how to install Talend Runtime Container as a service.
- Troubleshooting Talend ESB on page 81 describes how to fix problems related to JVM memory allocation parameters running Talend ESB.

Release directory structure

The Talend ESB software may be downloaded from https://www.talend.com/download/?utm_medium=help&utm_source=help_content.

The term <TalendRuntimePath> is used for the directory where Talend Runtime is installed. This is typically the full path of either Runtime_ESBSE or Talend-ESB-V, depending on the version of the software that is being used. Please substitute appropriately.

The Talend Runtime Container is in <TalendRuntimePath>/container, but if there are several containers, this may be actually <TalendRuntimePath>/container2, and so on. In this document, <RuntimeContainerPath> is used to indicate the related container path.

The standard directory layout of the Talend Runtime Container installation is as follows:

Starting/Stopping Talend Runtime and its ESB Infrastructure Components

This chapter describes how to start and stop Talend Runtime, the various options that are available, and how to start and stop the infrastructure components of Talend ESB.

Starting Talend Runtime

After starting Talend Runtime, you need to wait a few seconds for initialization to complete before entering the commands. Karaf starts the non core bundles in the background. So even if the console is already available, the job commands may not.

Warning: Do not run the container as root or as a privileged user for security reasons.

Starting Talend Runtime with console

About this task

From a console window:

Procedure

1. Change to the installation directory:

```
cd <RuntimeContainerPath>
```

2. Run the Talend Runtime:
   - `bin\trun.bat` (Windows)
   - `bin\trun` (Linux)

Results

Warning: Do NOT close the console or shell in which Karaf was started, as that will terminate Karaf (unless Karaf was started using the `nohup` command).

Starting Talend Runtime in the background

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

- `bin\start.bat` (Windows)
- `bin\start` (Linux)

Karaf can be reset to a clean state by simply deleting the `<RuntimeContainerPath>/data` folder, or by adding `clean` after the above `start` command.

Stopping Talend Runtime

Within the Karaf console, you can perform a shutdown using the `Ctrl+D` keys or the following `logout` command.

Within the Karaf console, to perform a clean shutdown of Karaf, use the `system:shutdown` command, with an optional `-f` (force) flag to avoid a `yes/no` confirmation.

Furthermore, from a command shell, you can run `bin\stop.bat` (Windows) or `bin/stop` (Linux).

For more advanced functionality, it is also possible to delay the shutdown using time arguments. The time arguments can have different formats:

- absolute time format `hh:mm`, in which `hh` is the hour (1 or 2 digits) and `mm` is the minute of the hour (in two digits).

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

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

- duration format `+m`, in which `m` is the number of minutes to wait. The argument `now` is an alias for `+0`.

  For example, the following command will shutdown Karaf in 10 minutes:

  ```
  system:shutdown +10
  ```

Starting and stopping Talend ESB infrastructure services

Talend Runtime provides infrastructure services, including the Service Locator, Service Activity Monitoring, Security Token Service for its community version, and additional XKMS, Service Registry, Authorization, and Event Logging, etc. for its subscription versions, which are already installed in the Talend Runtime Container.

To start all the infrastructure services, after starting the Talend Runtime Container, enter the following command at the console prompt:

```
tesb:start-all
```

Note: The Event Logging feature (subscription feature only) will not be automatically started by the `tesb:start-all` command. To start it, you will have to execute its specific command(s). For more information on how to start Event Logging, see the *Talend ESB Infrastructure Services Configuration Guide*.

You can shutdown all Talend ESB infrastructure services by entering:

```
tesb:stop-all
```

Note: If there is another Talend Runtime Container running on the same machine and you forgot to adapt the configuration of the used ports, a port conflict may arise while starting an infrastructure component. In this case `tesb:start-all` will not try to start the remaining infrastructure components. Stop all infrastructure services using `tesb:stop-all` and restart them after adapting the configuration. There are configuration scripts simplifying this task.

For more information on how to start and stop and how to configure each of the infrastructure services, see the *Talend ESB Infrastructure Services Configuration Guide*.

Using the console commands

To see a list of the available commands in the Talend Runtime Container(or Karaf) console press the **tab** at the prompt.

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

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

or

This chapter gives a summary of the commands, grouped by scope; help for each command is also available online at the Apache Karaf website.

Config Scope

The commands within the `config:` scope are used to modify configuration data of Karaf instances (data stored in the `/etc` folder of the Karaf instance.) Updates take effect immediately without need of restarting Karaf. Note there is also a ConfigMBean that can be used to do this configuration within JMX.

Table 1: config Scope commands

Command	Parameters	Description
`config:cancel`		Cancels the changes to the configuration being edited.
`config:delete`	`[options] pid`	Deletes a configuration.
`config:edit`	`[options] pid`	Creates or edits a configuration.
`config:list`	`[query]`	Lists existing configurations.
`config:meta`	`[options]`	Lists meta type information.
`config:property-append`	`[options] name value`	Appends the given value to an existing property or creates the property with the specified name and value.
`config:property-delete`	`[options] property`	Deletes a property from the configuration being edited.
`config:property-list`	`[options]`	Lists properties from the currently edited configuration.
`config:property-set`	`[options] property value`	Sets a property in the currently edited configuration.
`config:update`		Saves and propagates changes from the configuration being edited

Dev Scope

The `dev:` scope only has one command: dev:dump-create. You can use it to create zip archive with diagnostic info as follows:

```
dev:dump-create [options] [name]
```

Feature Scope

The commands within the `feature:` scope are used to provide support for Karaf features, which are predefined collections of bundles used to implement specific services.

Table 2: feature Scope commands

Command	Parameters	Description
`feature:info`	`[options] name [version]`	Shows information about selected feature.
`feature:install`	`[options] feature`	Installs a feature with the specified name and version.
`feature:list`	`[options]`	Lists all existing features available from the defined repositories.
`feature:regions`	`[options] [regions]`	Prints information about region digraph. Where `regions` are regions to provide detailed info for.
`feature:repo-add`	`[options] name/url [version]`	Adds a features repository. Where: • `name/url` is the shortcut name of the features repository or the full URL, • `version` is the version of the features repository if using features repository name as first argument. It should be empty if using the URL
`feature:repo-list`	`[options]`	Displays a list of all defined repositories.
`feature:repo-refresh`	`[Feature name or uri] [Feature version]`	Refresh a features repository. Where: • `Feature name or uri` is a shortcut name of the feature repository or the full URI • `Feature version` is the version of the feature if using the feature name. Should be empty if using the uri
`feature:repo-remove`	`[options] repository`	Removes the specified repository features service.
`feature:requirement-add`	`[options] requirements`	Adds provisioning requirements.
`feature:requirement-list`	`[options]`	Lists provisioning requirements.
`feature:requirement-remove`	`[options] requirements`	Removes provisioning requirements.

Command	Parameters	Description
`feature:start`	`[options] feature`	Starts features with the specified name and version.
`feature:stop`	`[options] feature`	Stops features with the specified name and version.
`feature:uninstall`	`[options] features`	Uninstalls a feature with the specified name and version.
`feature:version-list`	`[options] feature`	Lists all versions of a feature available from the currently available repositories.

JAAS Scope

The commands within the `jaas:` scope are used for management of JAAS users and realms.

Table 3: jaas Scope commands

Command	Parameters	Description
`jaas:cancel`		Cancels the modification of a JAAS realm.
`jaas:group-add`	`username group`	Makes a user part of a group.
`jaas:group-create`	`group`	Creates a group in a realm.
`jaas:group-delete`	`username group`	Removes a user from a group.
`jaas:group-list`		Lists groups in a realm.
`jaas:group-role-add`	`group role`	Adds a role to a group.
`jaas:group-role-delete`	`group role`	Removes a role from a group.
`jaas:pending-list`		Lists the pending modification on the active JAAS Realm/Login Module.
`jaas:realm-list`	`[options]`	Lists JAAS realms.
`jaas:realm-manage`	`[options]`	Manages user and roles of a Jaas Realm.
`jaas:role-add`	`username role`	Adds a role to a user.
`jaas:role-delete`	`username role`	Deletes a role from a user.
`jaas:su`	`[options] [user]`	Substitutes user identity.
`jaas:sudo`	`[options] [command]`	Executes a command as another user.

Command	Parameters	Description
`jaas:update`		Applies pending modification on the edited JAAS Realm.
`jaas:user-add`	`username password`	Adds a user.
`jaas:user-delete`	`username`	Deletes a user.
`jaas:user-list`	`[options]`	Lists the users of the selected JAAS realm/login module.

Log Scope

The commands within the `log:` scope are used for management of system logs.

Table 4: log Scope commands

Command	Parameters	Description
`log:clear`		Clears log entries.
`log:display`	`[options] [logger]`	Displays log entries.
`log:exception-display`	`[logger]`	Displays the last occurred exception from the log.
`log:get`	`[options] [logger]`	Shows the currently set log level.
`log:log`	`[options] message`	Logs a message.
`log:set`	`level [logger]`	Sets the log level.
`log:tail`	`[options] [logger]`	Continuously displays log entries. Use Ctrl+C to quit this command.

Shell Scope

The commands within the `shell:` scope are used to provide terminal window commands in the OSGi shell.

Table 5: Shell Scope commands

Command	Parameters	Description
`shell:alias`	`command`	Creates an alias to a command.
`shell:cat`	`[options] [paths or urls]`	Displays the content of a file or URL. Where `paths or urls` is a list of file paths or urls to display separated by whitespace (use - for STDIN)
`shell:clear`		Clears the console buffer.

Command	Parameters	Description
`shell:completion`	`[mode]`	Displays or changes the completion mode on the current console session. Where `mode` is the completion mode to set. The valid completion modes are: `global`, `first`, `subshell`.
`shell:date`	`[options] [format]`	Displays the current time in the given FORMAT.
`shell:each`	`values function`	Executes a closure on a list of arguments. Where: • `values` is the collection of arguments to iterate on • `function` is the function to execute
`shell:echo`	`[options] [arguments]`	Echoes or prints arguments to STDOUT, separated by whitespaces.
`edit`		
`shell:env`	`variable [value]`	Gets/sets the value of a console session variable.
`shell:exec`	`command`	Executes system processes.
`shell:grep`	`[options] pattern`	Prints lines matching the given pattern, in regular expression.
`shell:head`	`[options] [paths or urls]`	Displays the first lines of a file. Where `paths or urls` is a list of file paths or urls to display separated by whitespaces.
`shell:history`		Prints commands history.
`shell:if`	`condition ifTrue [ifFalse]`	If/Then/Else block.
`shell:info`		Prints system information.
`shell:java`	`[options] className [arguments]`	Executes a Java standard application.
`less`		
`shell:logout`		Disconnects shell from current session.
`more`		
`shell:new`	`class [args]`	Creates a new java object.
`shell:printf`	`format arguments`	Formats and prints arguments.
`shell:sleep`	`[options] duration`	Sleeps for a defined `duration` then wakes up. The default time unit is millisecond, use `-s` option to use second instead.
`shell:sort`	`[options] [files]`	Writes sorted concatenation of all files to standard output, separated by whitespaces.

Command	Parameters	Description
`shell:source`	`script [args]`	Runs a script. Where: • `script` is a URI pointing to the script • `args` are arguments for the script
`shell:stack-traces-print`	`[print]`	Prints the full stack trace in the console when the execution of a command throws an exception.
`shell:tac`	`[options]`	Captures the STDIN and returns it as a string. Optionally writes the content to a file.
`shell:tail`	`[options] [path or url]`	Displays the last lines of a file. Where `path or url` is a file path or url to display.
`shell:threads`	`[options] [id]`	Prints the current threads (optionally with stacktraces).
`shell:watch`	`[options] command`	Watches & refreshes the output of a command.
`shell:wc`	`[options] [files]`	Prints newline, word, and byte counts for each file.
`shell:while`	`condition function`	Loops while the condition is true.Where: • `condition` is the condition of the loop • `function` is the function to execute

Miscellaneous Scopes

There are a few scopes that offer just one or two commands each. They are listed in the below table.

Table 6: Miscellaneous Scope commands

Command	Parameters	Description
`package:exports`	`[options]`	Lists exported packages and the bundles that export them.
`package:imports`	`[options]`	Lists imported packages and the bundles that import them.
`ssh:ssh`	`[options] hostname [command]`	Connects to a remote SSH server.
`wrapper:install`	`[options]`	Installs the container as a system service in the OS.

Using datasources and connection pooling in Talend Runtime

If you are using a database connection component in one of your Jobs developed in the Talend Studio and want to deploy it on the Talend Runtime Container, before deploying it, you need to make sure the corresponding datasource is installed into that container.

This chapter describes how to setup the JDBC Drivers and how to install datasources in the Talend Runtime Container, and how to use datasources and connection pooling in the Talend Runtime.

Installing JDBC drivers into the Talend Runtime Container

As the Talend ESB package only provides the JDBC drivers for the Derby database, if you are using another database, you need to install its corresponding JDBC driver into the Talend Runtime Container before installing the datasource.

There are several ways to install the JDBC driver:

- Installing using a simple copy to the deploy folder on page 16,
- Installing the JDBC driver from a public Maven repository on page 16,
- Installing the JDBC driver from a local Maven repository on page 16,
- Installing the driver from the file system using bundle:install on page 17.

Then, to install the datasource, see Installing the DataSource in an OSGi container on page 18.

Installing using a simple copy to the deploy folder

Install the corresponding JDBC driver by copying the JDBC driver to the `<TalendRuntimePath>/deploy` folder.

Installing the JDBC driver from a public Maven repository

Since MySQL and H2 drivers are available in public repositories, they can be installed in one step using an `bundle:install` command in the container.

Here are the installation instructions for each of these (change the database version numbers if applicable):

- MySQL:

```
bundle:install mvn:mysql/mysql-connector-java/5.1.18
```

- H2:

```
bundle:install -s mvn:com.h2database/h2/1.3.165
```

Installing the JDBC driver from a local Maven repository

About this task

If there is no access to a public repository, the driver needs to be previously installed into a local repository. This is also true for proprietary databases such as Oracle, DB2 and SQLServer, as they do not publish their drivers in a public Maven repository.

Explicitly install the driver into local repository:

- either a Nexus repository (if your container is configured to work with Nexus),
- or a local repository accessible from the container.

Procedure

1. Install the driver into a repository using `mvn install`:
   - Oracle:

```
mvn install:install-file -Dfile=
    "C:\oraclexe\app\oracle\product\11.2.0\server\jdbc\lib\ojdbc6.jar"
    -DgroupId=ojdbc -DartifactId=ojdbc -Dversion=11.2.0.2.0
    -Dpackaging=jar
```

   - DB2:

```
mvn install:install-file
    -Dfile="C:\Program Files(x86)\IBM\SQLLIB\java\db2jcc.jar"
    -DgroupId=com.ibm.db2.jdbc -DartifactId=db2jcc -Dversion=9.7
    -Dpackaging=jar
```

   - SQLServer:

```
mvn install:install-file
    -Dfile="C:\sqljdbc4-3.0.jar"
    -DgroupId=com.microsoft.sqlserver -DartifactId=sqljdbc4
    -Dversion=3.0 -Dpackaging=jar
```

   - PostgreSQL:

```
mvn install:install-file
    -Dfile="C:\postgresql.jar"
    -DgroupId=postgresql -DartifactId=postgresql
    -Dversion=9.1-901.jdbc4 -Dpackaging=jar
```

2. Install the driver from the repository into a Talend Runtime Container using `bundle:install`:
   - Oracle:

```
bundle:install wrap:mvn:ojdbc/ojdbc/11.2.0.2.0
```

   - DB2:

```
bundle:install wrap:mvn:com.ibm.db2.jdbc/db2jcc/9.7
```

   - SQLServer:

```
bundle:install wrap:mvn:com.microsoft.sqlserver/sqljdbc4/3.0
```

   - PostgreSQL:

```
bundle:install wrap:mvn:postgresql/postgresql/9.1-901.jdbc4
```

Installing the driver from the file system using bundle:install

This is particularly useful for DB2, Oracle and SQLServer drivers since they are not published as OSGi bundles.

- Oracle:

```
bundle:install
wrap:file:E:/talend/TESB/db/oracle/ojdbc6.jar\\$Bundle-SymbolicName=
oracle.jdbc&Bundle-Version=11.2.0.2&Bundle-Name='JDBC Driver for Oracle'
```

- DB2:

```
bundle:install
wrap:file:E:/talend/TESB/db/db2/db2jcc-9.7.jar\\$Bundle-SymbolicName=
com.ibm.db2.jdbc&Bundle-Version=9.7&Bundle-Name='JDBC Driver for IBM DB2'
```

- SQLServer:

```
bundle:install
wrap:file:E:/talend/TESB/db/mssql/sqljdbc4-3.0.jar\\$Bundle-SymbolicName=
com.microsoft.sqlserver.jdbc&Bundle-Version=3.0&Bundle-Name=
'JDBC Driver for SQL Server'
```

- PostgreSQL:

```
bundle:install
wrap:file:E:/talend/TESB/db/postgresql/postgresql.jar\\$Bundle-SymbolicName
=postgresql&Bundle-Version=9.2&Bundle-Name=
'JDBC Driver for PostgreSQL'
```

Installing the DataSource in an OSGi container

About this task

Once the corresponding JDBC driver installed in the Talend Runtime Container, type in the following command on the console:

```
feature:install tesb-datasource-<Database>
```

The corresponding DataSource will be installed into the container and a configuration file named `org.talend.esb.datasource.<Database>.cfg` will be created in the `<Talend.runtime.dir>/container/etc` folder.

For example, to install the Derby DataSource:

Procedure

1. Execute the following command:

```
feature:install tesb-datasource-derby
```

2. On the Talend Runtime Container console, execute the `list` command, you will find the installed bundles and configuration of Derby driver:

```
[225] [Active] [       ] [ ] [60] Apache Derby 10.8 (10.8.1000002.
1095077)
[226] [Active] [Created] [ ] [60] Service Activity Monitoring ::
 Datasource-derby ()
```

The `org.talend.esb.datasource.derby.cfg` configuration file has been created into the `<Talend.runtime.dir>/container/etc` folder. In this configuration file, the Database

settings can be configured dynamically. For example, the default properties of `org.talend.esb.datasource.derby.cfg` are:

```
datasource.server=localhost
datasource.port=1527
datasource.database=db
datasource.createdatabase=create
datasource.user=test
datasource.password=test
```

Results

Here is a table with the DataSource information for other databases which work with the Talend Runtime Container:

DataSource Name	Database	Database, Driver Version	Feature	ConfigFile
ds-derby	Derby	10.8, 10.8.1.2	tesb-datasource-derby	org.talend.esb.datasource.derby.cfg
ds-h2	H2 Engine	1.3, 1.3.165	tesb-datasource-h2	org.talend.esb.datasource.h2.cfg
ds-mysql	MySQL	5.1, 5.1.18	tesb-datasource-mysql	org.talend.esb.datasource.mysql.cfg
ds-oracle	Oracle	11.2.0, 11.2.0.2.0	tesb-datasource-oracle	org.talend.esb.datasource.oracle.cfg
ds-db2	IBM DB2	9.7, 9.7	tesb-datasource-db2	org.talend.esb.datasource.db2.cfg
ds-sqlserver	SQL Server	2008R2, 3.0	tesb-datasource-sqlserver	org.talend.esb.datasource.sqlserver.cfg
ds-postgresql	PostgreSQL	9.2	tesb-datasource-postgresql	org.talend.esb.datasource.postgresql.cfg

Note: Other driver versions may work but have not been tested. For more information on software prerequisites, see the *Talend Installation and Upgrade Guide* .

Using datasources and connection pooling in Talend Runtime

About this task

Once the driver installed into the Talend Runtime Container(for more information, see Installing JDBC drivers into the Talend Runtime Container on page 16), you will be able to use the datasource in your database component in your Talend Job to deploy it in the Talend Runtime Container.

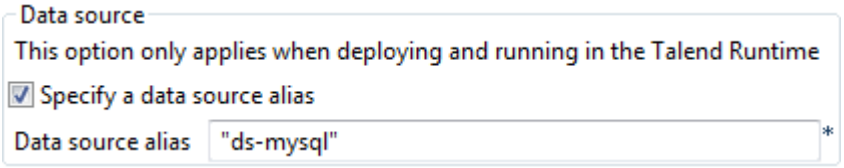
Procedure

1. Use the **DataSource Name** defined in the table below in the **Data source alias** field of your database connection component:

DataSource Name	Database
ds-derby	Derby
ds-h2	H2 Engine
ds-mysql	MySQL
ds-oracle	Oracle
ds-db2	IBM DB2
ds-sqlserver	SQL Server
ds-postgresql	PostgreSQL

For example, for the MySQL datasource:

Example



2. Now you can deploy your Job into the Talend Runtime Container.

Configuring XA transactions support in Talend ESB

Java Transaction API (JTA) allows transactions to span more than one resource. So, for example, a transaction can span two databases, or a database and a JMS Server.

The main integration point is the interface `javax.transaction.TransactionManager`. It allows to manage transactions as well as to enlist transactional resources. Only resources that are enlisted with a transaction are taking part in transactions coordinated by it.

The main interface for user code is `javax.transaction.UserTransaction`. It allows to begin, commit and rollback transactions but not to enlist resources.

See also the JTA 1.2 API.

Transaction Resources

The following technologies support XA transactions.

- JMS (ActiveMQ)
- JDBC (H2, Derby, Oracle, MySQL, PostgreSQL)
- JCR repositories (Jackrabbit)

Wrapping a DataSource for XA

An XA capable database offers an XADataSource. This data source is not to be used directly by users though. The user will always bind to the plain DataSource interface. It is the duty of the application server to wrap an XADataSource with pooling and XA auto-enlistment.

In OSGi, there is also the issue that standard JDBC does not work well with OSGi classloading. So the OSGi alliance specified the DataSourceFactory interface. A DataSourceFactory is provided as an OSGi service by the database driver or by an external adapter. It allows to create DataSource and XADataSource instances in an OSGi friendly way.

The pax-jdbc project provides DataSourceFactory adapters for legacy drivers as well as pooling and XA auto-enlistment wrappers for existing DataSourceFactory services.

For H2, the installation is below. This installs the H2 database as well as adapters for pooling and XA support.

```
feature:repo-add pax-jdbc 0.7.0
```

```
feature:install pax-jdbc-h2 pax-jdbc-pool-dbcp2 pax-jdbc-config transaction
```

The installation can be checked with `service:list` DataSourceFactory. It should list the original H2 DataSourceFactory as well as a pooling one and a pooling/XA one. The different services can be discerned by their service properties.

To create an actual DataSource, `pax-jdbc-config` monitors configuration files with the naming scheme `etc/org.ops4j.datasource-*.cfg`. It will automatically create a DataSource for each of these configurations.

You can reference the DataSourceFactory using the special property `osgi.jdbc.driver.name` or `osgi.jdbc.driver.class`. The value should match the respective property value of the DataSourceFactory service. For more details, see the pax-jdbc documenation.

XA support in Apache Aries JPA

Prerequisite is to have a correctly wrapped DataSource or DataSourceFactory like described above.

The next thing is to create a bundle for the JPA Entities. For the most part, this works like in JEE. The entities need to be correctly annotated and a `persistence.xml` should be put into `META-INF/persistence`. The main thing to keep in mind for OSGi is to add a special Manifest header Meta-Persistence: `META-INF/persistence.xml` which marks the bundle as containing a persistence unit and points to the location of the `persistence.xml`.

Aries JPA will monitor all bundles in state **Starting** and **Active** for the Meta-Persistence header above. It will then scan the `persistence.xml` and create a matching `EntityManagerFactory` and also an `EntityManager` as an OSGi service.

These services can then be leveraged in a second bundle that uses JPA to access the entities. The easiest way to do this is to leverage the blueprint-maven-plugin. It scans the classes for CDI/JEE annotations and creates a blueprint xml from them. The skeleton of a class using JPA would look like this:

```
@Singleton
@Transactional
public class BookRepositoryImpl {
    @PersistenceContext(unitName="ebook")
    EntityManager em;
}
```

`@Singleton` marks the class as a bean to be listed in the `blueprint.xml`. This is the major difference to CDI where it is not needed to mark beans in a special way.

`@Transactional` can be used on class or method level. An annotation on method level overrides the same on class level. By default, this will define a transaction of type `TxType.Required`. This is the typical type for methods that change JPA entities. It defines that the method will join a transaction if one exists, or create a new one if needed. For methods that only read data, the `TxType.Supports` can be used. It allows the method to participate in a transaction but also to run without. For a full description of the annotation, set the JTA 1.2 specification.

`@PersistenceContext` defines that the variable should be injected with the `EntityManager` of the named persistence unit. Aries JPA takes care of creating thread safe `EntityManager` that can be used like in JEE. The lifecycle of the `EntityManager` is bound to an OSGi Coordination. By default, the coordination spans that outermost call marked with `@Transactional`. This is important as the `EntityManager` itself is not thread safe, so Aries JPA internally needs to bind one instance to a thread but also needs to make sure the `EntityManager` is relatively short lived to avoid class loading problems in case of bundle restarts.

The user can also create a Coordination manually using the `CoordinatorService`. In the ebook example, this is done using a CXF interceptor that creates the coordination on the CXF level when a request is received and only ends it when the serialization is done. This makes the `EntityManager` available during serialization and allows to lazily load entities without the problem of having detached entities.

XA support and JMS in Apache Camel

It is also possible to use transactions in Apache Camel routes. Camel relies in the spring transaction abstraction so it is necessary to use the right wrapper class to make JTA transactions work.

A transactional route can look like this:

```
from("jms:queue1").transactional().to("jms:queue2")
```

The transactional() DSl Element tells Karaf to start a transaction when the route enters the segment and commit once the route is completed successfully. In case of unhandled errors, it will roll back.

The above approach would not cover the part through where the message is received. So to make the whole route transactional, it is necessary to also configure the JMS component to do JTA transactions.

For this to work like in databases, you first need to provide a `ConnectionFactory` as an OSGi service that is correctly wrapped for XA and pooling.

The `ConnectionFactory` and the `TransactionManager` are referenced as OSGi services. They then need to be injected into the `JMSComponent` and the `SpringTransactionPolicy`.

```
<reference id="connectionFactory" interface="javax.jms.ConnectionFactory"/>
<reference id="transactionManager" interface="org.springframework.transaction
.PlatformTransactionManager"/>

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="transactionManager" ref="transactionManager"/>
</bean>

<bean id="PROPAGATION_REQUIRED" class="org.apache.camel.spring.spi.SpringT
ransactionPolicy">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>
</bean>
```

With this set-up, the `JmsComponent` will always receive messages inside a JTA transaction and will also participate in a JTA transaction when sending out messages.

Interestingly, the transaction manager is looked up as spring `PlatformTransactionManager`. This is possible as the default transaction manager in Karaf from Aries already implements this interface in addition to the plain JTA TransactionManager.

Redelivery in JMS

Apache Camel can do redelivery using the camel error handling. This is not related to JTA though. So the more solid way is to configure redelivery in the JMS broker.

ActiveMQ can configure redelivery on the ConnectionFactory and centrally on the broker. For more information, see: http://activemq.apache.org/message-redelivery-and-dlq-handling.html. The user can set the number of redeliveries, the backoff behaviour and the dead letter queue where messages end up if all fails.

Deploying Multiple Karaf Containers

This chapter describes how to adapt the configuration of the container(s) to deploy several of them on the same machine.

Deploying multiple containers using configuration adaption scripts

About this task

In order to avoid conflicts between multiple container instances, there are Karaf configuration adaption scripts which automate the adjustment of potentially conflicting parameters. They are based on the `edit` and `property-set` commands described above.

Those scripts are executed within the Karaf container and thus are supported by all platforms compatible with container. So, even if they have a ".sh" extention, their usage is not limited to Linux based operating systems.

To deploy multiple containers:

Procedure

1. Make sure that the default container you will copy has never been started before. Or if it has already been started, make sure it is stopped (using the `system:shutdown` or `logout` commands or **Ctrl+D**), and that its `data` directory has been deleted, before copying its directory and files to create containers 1, 2 and 3. Otherwise, the data in the default container (which contains absolute paths relating to the default container) will cause problems in the others.
2. In a Linux terminal or Windows command line window, change directory to `<TalendRuntimePath>`.
3. Copy the directory structure of the default container with all its content to produce the first container by executing the most applicable command:

   - `cp -r container container1` (Linux)
   - `robocopy /e container container1` (Windows)
   - `xcopy /e container container1` (older Windows with no robocopy)

   Now, you have a second container in directory `container1`.

   Repeat the above command to create the second and the third containers in the same way.
4. Start the first container:

   - Under Linux, go to directory `<TalendRuntimePath>/container1` and execute the following command:

     ```
     ./bin/trun
     ```
   - Under Windows, go to directory `<TalendRuntimePath>\container1` and execute the following command:

     ```
     .\bin\trun.bat
     ```

   Start the second and the third containers in the same way.

5. When the initialization is complete, run the configuration adaption script at the container prompt to update and save the new settings:
   - `source scripts/configureC1.sh` for the first container copy
   - `source scripts/configureC2.sh` for the second container copy
   - `source scripts/configureC3.sh` for the third container copy
   - `source scripts/configureC0.sh` resets the parameters to the default values

   All necessary parameters adjustments are done using this single script call. Changes performed by the configuration adaption scripts are persistent and reflected in the configuration files in `container/etc`.
6. Shut down and restart the container after applying a configuration adaption script, to make sure the new parameters are used. Most of the parameter changes will be adapted "on the fly", but for the Jobserver parameters in Talend ESB this is not yet possible.

Results

The ports which are configured using the configuration scripts are described in the table below:

Parameter	configureC0.sh	configureC1.sh	configureC2.sh	configureC3.sh
HTTP Port	8040	8041	8042	8043
HTTPS Port	9001	9002	9003	9004
RMI Registry Port	1099	1100	1101	1102
RMI Server Port	44444	44445	44446	44447
SSH Port	8101	8102	8103	8104
Command Port	8000	8010	8020	8030
File Transfer Port	8001	8011	8021	8031
Monitoring Port	8888	8898	8908	8918

Troubleshooting

If you get the "Port already in use exception" when starting an alternate container, recheck that there is not already a container running using the default parameters.

If you are still getting the error, it may also be that the port is actually in use by an unrelated process, so change the ports in the configuration adaption scripts and rerun these to apply the changes.

Remote Console

It does not always make sense to manage an instance of Talend ESB using the local console. Talend ESB can be remotely managed using a remote console.

When you start Karaf, it enables a remote console that can be accessed over SSH from any other Karaf console or plain SSH client. The remote console provides all the features of the local console and gives a remote user complete control over the container and services running inside of it.

Configuring remote instances

The SSH hostname and port number of the remote console is configured in the `<RuntimeContainerPath>/etc/org.apache.karaf.shell.cfg` configuration file with the following defaults values:

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

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

```
# define helper functions
bundle-by-sn = { bm = new java.util.HashMap ; //
   each (bundles) { $bm put ($it symbolicName) $it } ; $bm get $1 }
bundle-id-by-sn = { b = (bundle-by-sn $1) ;  //
   if { $b } { $b bundleId } { -1 } }

# edit config
config:edit org.apache.karaf.shell
config:property-set sshPort 8102
config:update

# force a restart
bundle:restart --force (bundle-id-by-sn org.apache.karaf.shell.ssh)
```

Connecting and disconnecting remotely

Using the ssh:ssh command

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

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

Warning: The default password is tadmin but we recommend changing it. See Security on page 33 for more information.

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

```
Karaf
  Karaf home                  /local/apache-karaf-2.0.0
  Karaf base                  /local/apache-karaf-2.0.0
  OSGi Framework              org.eclipse.osgi - 3.5.1.R35x_v20090827
JVM
```

```
  Java Virtual Machine         Java HotSpot(TM) Server VM version 14.1-b02
  ...
```

Using the Karaf client

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

For example, to quickly connect to a Karaf instance running in server mode on the same machine, run the following command from the karaf-install-dir directory: `bin/client`. More usually, you would provide a hostname, port, username and password to connect to a remote instance. And, if you were using the client within a larger script, you could append console commands as follows:

```
bin/client -a 8101 -h hostname -u tadmin -p tadmin feature:install wrapper
```

To display the available options for the client, type:

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

Using a plain SSH client

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

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

Disconnecting from a remote console

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

Stopping a remote instance

Using the remote console

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

> **Note:** Pressing Ctrl+D in a remote console simply closes the remote connection and returns you to the local shell without shutting off the remote instance.

Using the Karaf client to stop a remote instance

To stop a remote instance using the Karaf client, run the following from the `karaf-install-dir/lib` directory:

```
bin/client -u tadmin -p tadmin -a 8101 -h hostname system:shutdown
```

Apache CXF and Camel, commands and tuning

These are commands that are related to Apache CXF and Camel functionality.

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

Commands supplied by CXF

These commands related to CXF functionality. For more information on CXF see http://cxf.apache.org/.

Table 7: CXF commands

Command	Parameters	Description
`cxf:list-busses`		Lists all CXF Busses
`cxf:list-endpoints`	`[options] [bus]`	Lists all CXF Endpoints on a Bus. Where `bus` is the optional CXF bus name where to look for the Endpoints
`cxf:stop-endpoint`	`bus endpoint`	Stops a CXF Endpoint on a Bus. Where: • `bus` is CXF bus name where to look for the Endpoint. • `endpoint` is the Endpoint name to stop.
`cxf:start-endpoint`	`[options] bus endpoint`	Starts a CXF Endpoint on a Bus. Where: • `bus` is CXF bus name where to look for the Endpoint. • `endpoint` is the Endpoint name to start.

Commands supplied by Camel

These commands related to Camel functionality. Help for these commands is available at http://camel.apache.org/karaf.html. These are for version Camel , which is the current version used by Talend ESB.

Note: Use TAB key for completion on the name parameters below.

Table 8: Camel commands

Command	Parameters	Description
`camel:component-list`	`[options] name`	Lists all Camel components that are in use in Karaf.
`camel:context-list`		Lists the Camel contexts available in the current Karaf instance

Command	Parameters	Description
`camel:context-inflight`	`[options] name`	Lists inflight exchanges.
`camel:context-info`	`[options] name`	Displays detailed information about a given Camel context
`camel:context-resume`	`context`	Resumes a Camel context.
`camel:context-start`	`context`	Starts the given Camel context.
`camel:context-stop`	`context`	Stops the given Camel context. It becomes unavailable and can not be started again.
`camel:context-suspend`	`context`	Suspends a Camel context.
`camel:eip-explain`	`[options] name nameOrId`	Explains the EIP in the CamelContext. Where: • `name` is the name of the Camel context • `nameOrId` is the name of the EIP or a node id to refer to a specific node from the routes
`camel:endpoint-explain`	`[options] name`	Explains all Camel endpoints available in the CamelContext.
`camel:endpoint-list`	`[options] name`	Lists endpoints from all camel contexts available in the current Karaf instance.
`camel:rest-registry-list`	`[options] name`	Lists all Camel REST services enlisted in the Rest Registry from a CamelContext.
`camel:rest-show`	`context`	Display the Camel REST definition in XML.
`camel:route-list`	`[name]`	Displays the list of Camel routes available in the current Karaf instance, where `name` is the Camel context name where to look for the route.
`camel:route-info`	`[options] name`	Provides detail information about a Camel route.
`camel:route-profile`	`route [context]`	Displays information about a Camel route. Where: • `route` is the Camel route ID. • `context` is the Camel context name.
`camel:route-reset-stats`	`context`	Resets route performance stats from a CamelContext.
`camel:route-resume`	`route context`	Resumes the given route. Where: • `route` is the Camel route ID or a wildcard expression. • `context` is the Camel context name.
`camel:route-show`	`route [context]`	Displays the Camel route definition in XML.

Command	Parameters	Description
`camel:route-start`	`route [context]`	Starts the given route. Where: • `route` is the Camel route ID or a wildcard expression. • `context` is the Camel context name.
`camel:route-stop`	`route [context]`	Stops the given route. Where: • `route` is the Camel route ID or a wildcard expression. • `context` is the Camel context name.
`camel:route-suspend`	`route [context]`	Suspends the given route. Where: • `route` is the Camel route ID or a wildcard expression. • `context` is the Camel context name.

Configuring CXF workqueues

CXF workqueues are used for queuing incoming work requests using a thread pool.

The `etc/org.apache.cxf.workqueues.cfg` configuration file is used for workqueue configuration (without this file, the configuration would need to be done for each individual bundle using CXF services). This process can significantly optimize the performance of HTTP / CXF Service request handling in the Talend Runtime Container.

This mechanism allows you to configure global workqueues for use by all bundles that are created, and normally services would share a thread pool. However an individual service can override this via local configuration if they have specific requirements.

Usage of queues

Workqueue settings may be used in different scenarios:

- For SOAP-based web service providers, the ws-addressing workqueue (Configuration files on page 32) can be used, and the default workqueue for One-Way (no response) SOAP calls. (Workqueues are not applicable for REST endpoints.)
- For JMS transport, workqueues are active to handle continuations; for regular JMS, there are similar settings, like the number of consumers or the thread pool for execution, which are implemented outside of workqueues.
- SOAP clients can use the "http-conduit" workqueue for asynchronous calls.

Configuration parameters

A configuration file contains the following parameters:

Name	Description	Default value
org.apache.cxf.workqueue.names	One or more names of workqueues, separated by commas	'default'
org.apache.cxf.workqueue.default.highWaterMark	Maximum number of threads to work on the queue	10

Name	Description	Default value
org.apache.cxf.workqueue.default.lowWaterMark	Minimum number of threads to work on the queue	5
org.apache.cxf.workqueue.default.initialSize	Initial number of threads to work on the queue	7
org.apache.cxf.workqueue.default.dequeueTimeout	This is the keep alive time for the threadpool executor. This allows excess threads to be terminated when idle for longer than this time, and they can be created again later if needed.	100 (optional)
org.apache.cxf.workqueue.default.queueSize	Maximum number of entries in the queue	100 (optional)

Configuration files

In the Talend Runtime Container, the default configuration file is in `etc/org.apache.cxf.workqueues.cfg`.

Note this file is normally not used or edited directly; it gives workqueue default values that each CXF-using bundle can choose to employ when it creates its work queues.

To configure a workqueue, there is a specific corresponding file: `org.apache.cxf.workqueues-n.cfg`, where typically "n" is the same as the workqueue name; for example, `org.apache.cxf.workqueues-http-conduit.cfg` would configure the `http-conduit` workqueue. At the moment, this functionality is for pre-defined work queues; it is not possible to use it for user-defined workqueues.

Here is the list of pre-defined workqueue names:

Name	Description
default	this means using the default values.
http-conduit	On the client side when using the asynchronous methods, the HTTP conduit must wait for and process the response on a background thread. This can control the queue that is used specifically for that purpose, to limit or expand the number of outstanding asynchronous requests.
jms-continuation	This is used by the JMS transport to handle continuations.
local-transport	The local transport being based on PipedInput/OutputStreams requires the use of separate threads; this workqueue can be used to configure the queue used exclusively for the local-transport.
ws-addressing	For decoupled cases, the ws-addressing layer may need to process the request on a background thread. This can control the workqueue it uses.

Note: You can also update these variables (for example `org.apache.cxf.workqueue.default.initialSize`) using the standard Karaf configuration commands.

Security

This chapter provides the information you need to manage MDM users from Talend MDM Web UI.

Managing users and passwords

The default security configuration uses a property file located at `<RuntimeContainerPath>/etc/users.properties` to store authorized users and their passwords. The default user name is `tadmin` and the associated password is `tadmin` too. We strongly encourage you to change the default password by editing the above file before moving Karaf into production.

The users are currently used in three different places in Karaf:

- access to the SSH console
- access to the JMX management layer
- access to the Web console

Those three ways all delegate to the same JAAS based security authentication.

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

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

Managing roles

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

```
[classname:]principal
```

Where `classname` is the class name of the principal object (defaults to `org.apache.karaf.jaas.modules.RolePrincipal`) and `principal` is the name of the principal of that class (defaults to `admin`). Note that roles can be changed for a given layer using ConfigAdmin in the following configurations:

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

Deploying security providers

Some applications require specific security providers to be available, such as BouncyCastle. The JVM imposes some restrictions about the use of such provider JAR files, namely, that they need to be

signed and be available on the boot classpath. One way to deploy such providers is to put them in the JRE folder at `$JAVA_HOME/jre/lib/ext` and modify each provider's security policy configuration (`$JAVA_HOME/jre/lib/security/java.security`) in order to register them. While this approach works fine, it has a global effect and requires that all servers are configured accordingly.

However Talend ESB offers a simple way to configure additional security providers:

- put your provider jar in `[karaf-install-dir]/lib/ext`
- modify the `[karaf-install-dir]/etc/config.properties` configuration file to add the following property:

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

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

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

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

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

Enabling password encryption

In Talend ESB, most of the security relevant properties, like passwords, are stored as configuration files in `etc/*.cfg` files and are loaded by Karaf using the OSGI ConfigAdmin. To avoid storing them as clear text properties in Talend ESB, but also for CXF key stores and general property files not managed by OSGI, Talend provides encryption capabilities, based on Jasypt, to encrypt and decrypt them on the fly, before being used by the ESB components.

In the following sections, you will find how you can configure your Talend ESB Container to be able to encrypt and decrypt your security properties. Encrypted property values are generally wrapped in a `ENC()` function.

Encrypting clear text parameters and passwords

About this task

The Talend ESB Runtime provides a utility to enable you to encrypt your desired clear text parameters and passwords. You can there after use the encrypted values in your configuration and property files.

Follow these steps to use this utility:

Procedure

1. Set the environment variable `TESB_ENV_PASSWORD` in `esb/container/bin/setenv.bat` as follows:

   ```
   SET TESB_ENV_PASSWORD=pwd
   ```

   Where pwd can be anything you specified.

This is the master password used by the container to encrypt all the desired parameters and passwords.

2. Start the Talend Runtime Container.
3. Install the `tesb-encryptor-command` feature in the container:

```
karaf@trun()>feature:install tesb-encryptor-command
```

4. Get an encrypted string for 'tadmin' for example by entering the following command in the container:

```
karaf@trun()> tesb:encrypt-text tadmin
```

You will get the following result:

```
ENC(nfTSDFdyRe3QUDucWhzpOUDBQQsYQnKgqnTdy334bs4=)
```

In case the password is not specified, it will be taken from the system environment variable TESB_ENV_PASSWORD. This helper uses PBEWITHSHA256AND128BITAES-CBC-BC hardcoded algorithm.

Note that to use passwords encrypted by the `tesb-encryptor-command` feature in Data Services and Routes, you need to store the encrypted password in a context variable and specify the context variable in the password field of tESBConsumer, tRESTClient, cSOAP or cREST component, then deploy the service via Talend Administration Center and overwrite the context variable with the encrypted value. This is the only way encryption of passwords in Data Services and Routes works.

Warning: All versions of Java from 1.8.0_161 enable the unlimited strength policy files by default, see https://www.oracle.com/technetwork/java/javase/8u161-relnotes-4021379.html for more information. If JCE security extensions are missing from JRE/JDK installation, you may face the error message, "Error executing command: java.lang.SecurityException: JCE cannot authenticate the provider BC". Those extensions are typically not installed by default because of the US export restrictions placed on "strong cryptography". Download them at http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html for Java 8 or http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html for Java 7.

OSGI Configuration Parameters for Blueprint Components

About this task

As Jasypt supports blueprint components, it can be easily configured to use it.

Procedure

1. Add this namespace to the blueprint file:

```
xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0"
```

2. Configure the jasypt as follows:

```
<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="PBEWITHSHA256AND128BITAES-CBC-BC"/>
 <property name="passwordEnvName" value="TESB_ENV_PASSWORD"/>
 </bean>
 </property>
 </enc:encryptor>
</enc:property-placeholder>
```

3. Create the encrypted properties as described in Encrypting clear text parameters and passwords on page 34.
4. Add the encrypted property inside the `ENC()` prefix like in the following example:

```
<cm:property-placeholder persistent-id="my" update-strategy="reload">
    <cm:default-properties>
        <cm:property name="password" value="ENC(T4H3aL4AfR20Tl+S9krZQVzTFCVG9ak
KxWmfDAobPxM=)"/>
    </cm:default-properties>
</cm:property-placeholder>
```

5. If you are using Maven, the following dependencies should be added to the `pom.xml`:

```
<dependency>
    <groupId>org.apache.servicemix.bundles</groupId>
    <artifactId>org.apache.servicemix.bundles.jasypt</artifactId>
    <version>1.9.2_1</version>
</dependency>
<dependency>
    <groupId>org.apache.karaf.jaas.blueprint</groupId>
    <artifactId>org.apache.karaf.jaas.blueprint.jasypt</artifactId>
    <version>4.0.1</version>
</dependency>
```

6. Import the package `org.jasypt.encryption.pbe` into the bundle. If you are using Maven, the following instructions should be added to the `pom.xml`:

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.4.0</version>
  <extensions>true</extensions>
  <configuration>
     <instructions>
        <Import-Package>org.jasypt.encryption.pbe;version=1.9.2, org.jasypt.enc
ryption.pbe.config;version=1.9.2, org.osgi.service.blueprint</Import-Package>
     </instructions>
  </configuration>
</plugin>
```

7. Section `<cm:property-placeholder>` must be defined before (upward) section `<enc:property-placeholder>` inside the blueprint configuration, otherwise decryption of parameters from `etc/*.cfg` will not work.

Results

An example of blueprint configuration (environment variable TESB_ENV_PASSWORD is set to pwd) is shown below:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
   xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0"
   xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/blueprint/
v1.0.0/blueprint.xsd
      http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0 http://aries.a
pache.org/schemas/blueprint-cm/blueprint-cm-1.1.0.xsd">
<cm:property-placeholder persistent-id="my" update-strategy="reload">
 <cm:default-properties>
  <cm:property name="password" value="ENC(ri+N4zeF/hTl1omjgYky1uQxYwhyxyPmdnyC/UmY1ug=)
" />
 </cm:default-properties>
</cm: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="PBEWITHSHA256AND128BITAES-CBC-BC" />
    <property name="passwordEnvName" value="TESB_ENV_PASSWORD" />
   </bean>
  </property>
 </enc:encryptor>
</enc:property-placeholder>

<bean id="serviceBean" class="org.company.example.MyServiceImpl">
          <property name="prop" value="${password}"/>
</bean>

<blueprint>
```

An example of persistent configuration (`container/etc/my.cfg`) is as follows:

```
password=ENC(6laeC861kCMSh2Eaj4sjtNzgstdo5BKFh8d+fop2Jt0=)
```

OSGI Configuration Parameters for Spring Components

About this task

If you are using a Java client and using OSGi configuration parameters, then follow these steps.

To use jasypt with spring components:

Procedure

1. Add the following Maven dependencies:

```
<dependency>
  <groupId>org.apache.servicemix.bundles</groupId>
  <artifactId>org.apache.servicemix.bundles.jasypt</artifactId>
  <version>1.9.2_1</version>
</dependency>
<dependency>
  <groupId>org.apache.servicemix.bundles</groupId>
  <artifactId>org.apache.servicemix.bundles.jasypt-spring31</artifactId>
  <version>1.9.2_1</version>
</dependency>
```

2. Add the usual properties, but the encrypted property should be wrapped by the `ENC( )` function.

```
<osgix:cm-properties id="properties" persistent-id="my">
  <prop key="mydb.password">ENC(fAdXf/ed2k+2uD1LhoOpvw==)</prop>
</osgix:cm-properties>
```

3. Add the password and the algorithm configuration for encryptor.

```
<bean id="environmentVariablesConfiguration" class="org.jasypt.encryption
.pbe.config.EnvironmentStringPBEConfig">
   <property name="algorithm" value="PBEWITHSHA256AND128BITAES-CBC-BC"/>
   <property name="password" value="TESB_ENV_PASSWORD"/>
</bean>
```

4. Create the encryptor and inject configuration to it.

```
<bean id="configurationEncryptor" class="org.jasypt.encryption.pbe.StandardP
BEStringEncryptor">
   <property name="config" ref="environmentVariablesConfiguration"/>
</bean>
```

5. Add the property configurer and pass the properties to it. The `EncryptablePro pertyPlaceholderConfigurer` will read the `.properties` files and make their values accessible as `${var}`.

```
<bean id="propertyConfigurer"
 class="org.jasypt.spring31.properties.EncryptablePropertyPlaceholderCon
figurer">
   <constructor-arg ref="configurationEncryptor"/>
   <property name="properties" ref="properties"/>
</bean>
```

6. Now, you can use the properties as usual:

```
<bean id="serviceBean" class="org.company.example.MyServiceImpl">
    <property name="prop" value="${mydb.password}"/>
</bean>
```

Jasypt with Data Services

If you wish to encrypt the context parameters or passwords while using Data services, use the utility explained in Encrypting clear text parameters and passwords on page 34 and there after use the encrypted values in your context parameters.

Encrypting passwords in CXF crypto property files

About this task

Since CXF version 3.X, CXF uses Apache WSS4J 2.X which according to http://ws.apache.org/wss4j/migration/newfeatures20.html supports encrypting passwords in Crypto properties files using Jasypt.

In http://stackoverflow.com/questions/31023223/encrypting-passwords-in-crypto-property-files, a more detailed description can be found:

Procedure

1. Download the jasypt-1.9.2-dist.zip (or newer) from http://www.jasypt.org/download.html.
2. Get an Encoded password with this command `encrypt input=real_keystore_password password=master_password algorithm=PBEWithMD5AndTripeDES`
3. Copy the OUTPUT (For example: 0laAaRahTQJzlsDu771tYi)

4. As you are using this algorithm, you need the Java Cryptography Extension (JCE) Unlimited Strength in your JDK.
5. Put the encoded OUTPUT in the properties.

```
org.apache.wss4j.crypto.provider=org.apache.wss4j.common.crypto.Merlin
org.apache.wss4j.crypto.merlin.keystore.type=jks
org.apache.wss4j.crypto.merlin.keystore.password=ENC(0laAaRahTQJzlsDu771
tYi)
org.apache.wss4j.crypto.merlin.keystore.alias=my_alias
org.apache.wss4j.crypto.merlin.keystore.file=/etc/cert/my_keystore.jks
```

6. In the CallbackHandler, put the master_password that you used to generated the encoded one:

```
public class WsPasswordHandler implements CallbackHandler {
  @Override
  public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
    for (Callback callback: callbacks){
        WSPasswordCallback pwdCallback= (WSPasswordCallback) callback;
        final int usage=pwdCallback.getUsage();
        if (usage==WSPasswordCallback.SIGNATURE||usage==WSPasswordCallback.DECRYPT)
{
            pwdCallback.setPassword("parKeyPassword");
        }
        if (usage==WSPasswordCallback.PASSWORD_ENCRYPTOR_PASSWORD){
            pwdCallback.setPassword("master_password");
        }
    }
  }
}
```

Encrypting Jackrabbit Configuration

When using a database-backed persistence manager or another component, you usually need to include the database password in Jackrabbit configuration. If you do not want to store such passwords in plain text inside the configuration file, you can encode the password in base64 and prefix it with `{base64}`. Jackrabbit will automatically decode such a password before passing it to the underlying database.

Managing realms

Karaf supports JAAS with some enhancements to allow JAAS to work nicely in an OSGi environment. This framework also features an OSGi keystore manager with the ability to deploy new keystores or truststores at runtime.

Overview

The Security framework feature of Karaf allows runtime deployment of JAAS based configuration for use in various parts of the application. This includes the remote console login, which uses the karaf realm, but which is configured with a dummy login module by default. These realms can also be used by the NMR, JBI components or the JMX server to authenticate users logging in or sending messages into the bus.

In addition to JAAS realms, you can also deploy keystores and truststores to secure the remote shell console, setting up HTTPS connectors or using certificates for WS-Security.

A very simple XML schema for spring has been defined, allowing the deployment of a new realm or a new keystore very easily.

Schema

To override or deploy a new realm, you can use the following XSD which is supported by a Spring namespace handler and can thus be defined in a Spring xml configuration file.

You can find the schema at the following location: http://karaf.apache.org/xmlns/jaas/v1.1.0.

Here are two examples using this schema:

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

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

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

</blueprint>
```

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

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

Each realm can contain one or more module definitions. Each module identifies a LoginModule and the `className` attribute must be set to the class name of the login module to use. Note that this login module must be available from the bundle classloader, so either it has to be defined in the bundle itself, or the needed package needs to be correctly imported. The `flags` attribute can take one of four values that are explained on the JAAS documentation. The content of the `module` element is parsed as a properties file and will be used to further configure the login module.

Deploying such a code will lead to a JaasRealm object in the OSGi registry, which will then be used when using the JAAS login module.

Configuration override and use of the rank attribute

The `rank` attribute on the config element is tied to the ranking of the underlying OSGi service. When the JAAS framework performs an authentication, it will use the realm name to find a matching JAAS configuration. If multiple configurations are used, the one with the highest `rank` attribute will be used. So if you want to override the default security configuration in Karaf (which is used by

the ssh shell, web console and JMX layer), you need to deploy a JAAS configuration with the name `name="karaf"` and `rank="1"`.

```
<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 that allows the $[karaf.base] property to be resolved -->
   <ext:property-placeholder
      placeholder-prefix="$[" placeholder-suffix="]"/>

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

</blueprint>
```

Architecture

Due to constraints in the JAAS specification, one class has to be available for all bundles. This class is called ProxyLoginModule and is a LoginModule that acts as a proxy for an OSGi 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 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.

Available realms

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

PropertiesLoginModule

This login module is the one configured by default. It uses a properties text file to load the users, passwords and roles from. This file uses the properties file format. The format of the properties are as follows, each line defining a user, its password and the associated roles: `user=password[,role][,role]...`

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

OsgiConfigLoginModule

The OsgiConfigLoginModule uses the OSGi ConfigurationAdmin service to provide the users, passwords and roles. Instead of `users` for the PropertiesLoginModule, this configuration uses a `pid` value for the process ID of the configuration containing user definitions.

JDBCLoginModule

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

Name	Description
datasource	The datasource as on OSGi ldap filter or as JDNI name
query.password	The SQL query that retries the password of the user
query.role	The SQL query that retries the roles of the user

Passing a data source as an OSGi ldap filter

To use an OSGi ldap filter, the prefix osgi: needs to be provided. See the example below:

```
<jaas:config name="karaf">
   <jaas:module
      className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
      flags="required">
      datasource = osgi:javax.sql.DataSource/   \\
         (osgi.jndi.service.name=jdbc/karafdb)
      query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
      query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
   </jaas:module>
</jaas:config>
```

Passing a data source as a JNDI name

To use an JNDI name, the prefix jndi: needs to be provided. The example below assumes the use of Aries JNDI to expose services via JNDI.

```
<jaas:config name="karaf">
   <jaas:module
      className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
      flags="required">
      datasource = jndi:aries:services/javax.sql.DataSource/   \\
         (osgi.jndi.service.name=jdbc/karafdb)
         query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
         query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
   </jaas:module>
</jaas:config>
```

LDAPLoginModule

The LDAPLoginModule uses a LDAP to load the users and roles, bind the users on the LDAP to check passwords. The LDAPLoginModule supports the following parameters:

Name	Description
connection.url	The LDAP connection URL, e.g. ldap://hostname
connection.username	Admin username to connect to the LDAP. This parameter is optional, if it's not provided, the LDAP connection will be anonymous.
connection.password	Admin password to connect to the LDAP. Only used if the connection.username is specified.
user.base.dn	The LDAP base DN used to looking for user, e.g. ou=user,dc=apache,dc=org

Name	Description
user.filter	The LDAP filter used to looking for user, e.g. (uid=%u) where %u will be replaced by the username.
user.search.subtree	If "true", the user lookup will be recursive (SUBTREE). If "false", the user lookup will be performed only at the first level (ONELEVEL).
role.base.dn	The LDAP base DN used to looking for roles, e.g. ou=role,dc=apache,dc=org
role.filter	The LDAP filter used to looking for user's role, e.g. (member:=uid=%u)
role.name.attribute	The LDAP role attribute containing the role string used by Karaf, e.g. cn
role.search.subtree	If "true", the role lookup will be recursive (SUBTREE). If "false", the role lookup will be performed only at the first level (ONELEVEL).
authentication	Define the authentication backend used on the LDAP server. The default is simple.
initial.context.factory	Define the initial context factory used to connect to the LDAP server. The default is com.sun.jndi.ldap.LdapCtxFactory
ssl	If "true" or if the protocol on the connection.url is ldaps, an SSL connection will be used
ssl.provider	The provider name to use for SSL
ssl.protocol	The protocol name to use for SSL (SSL for example)
ssl.algorithm	The algorithm to use for the KeyManagerFactory and TrustManagerFactory (PKIX for example)
ssl.keystore	The key store name to use for SSL. The key store must be deployed using a jaas:keystore configuration.
ssl.keyalias	The key alias to use for SSL
ssl.truststore	The trust store name to use for SSL. The trust store must be deployed using a jaas:keystore configuration.

An example of LDAPLoginModule usage follows:

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

If you want to use an SSL connection, the following configuration can be used as an example:

```
<ext:property-placeholder />

<jaas:config name="karaf" rank="1">
   <jaas:module
   className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
   flags="required">
   connection.url = ldaps://localhost:10636
   user.base.dn = ou=users,ou=system
   user.filter = (uid=%u)
   user.search.subtree = true
   role.base.dn = ou=groups,ou=system
   role.filter = (uniqueMember=uid=%u)
   role.name.attribute = cn
   role.search.subtree = true
   authentication = simple
   ssl.protocol=SSL
   ssl.truststore=ks
   ssl.algorithm=PKIX
   </jaas:module>
</jaas:config>

<jaas:keystore name="ks"
   path="file:///${karaf.home}/etc/trusted.ks"
   keystorePassword="secret" />
```

HTTP Configuration

HTTP is a request and response protocol used to enable communications between clients and servers.

Basically, a client send a request to a server on a particular port. The server listening to that port is waiting for the client's request. And when it receives the request from the client, the server sends back the corresponding response.

To secure this communication, a SSL protocol can be used on top of HTTP to provide security. This will allow the data to be encrypted and safely transfered through a secure HTTP protocol: HTTPS.

Server HTTP Configuration

The Talend Runtime provides support for HTTP and HTTPS by default with the help of the pax web component. For more information, see the documentation of Pax Web on http://team.ops4j.org/wiki/display/paxweb/Documentation.

HTTP / HTTPS configuration for Talend Runtime is done in the following configuration file: `<TalendRuntimePath>/container/etc/org.ops4j.pax.web.cfg`.

Basic configuration

To enable HTTP and HTTPS in Talend Runtime, configure the properties of the `org.ops4j.pax.web.cfg` file as follows:

Property	Default	Description
`org.osgi.service.http.port`	8040	This property specifies the port used for servlets and resources accessible via HTTP. The default value for this property is 8040. You can specify a value of 0 (zero), if you wish to allow Pax Web to automatically determine a free port to use for HTTP access.
`org.osgi.service.http.port.secure`	9001	This property specifies the port used for servlets and resources accessible via HTTPS. The default value for this property is 9001. You can specify a value of 0 (zero), if you wish to allow Pax Web to automatically determine a free port to use for HTTPS access.
`org.osgi.service.http.enabled`	true	This property specifies if the HTTP is enabled or disabled. If the value is set to "true", the support for HTTP access is enabled. If the value is set to "false", the support for HTTP access is disabled. The default value is "true".
`org.osgi.service.http.secure.enabled`	false	This property specifies if the HTTPS is enabled. If the value is set to "true", the support for HTTPS access is enabled. If the value is set to "false", the support for HTTPS access is disabled. The default value is "false".

SSL configuration

To encrypt the communication and secure the identification of a server, you can use a HTTPS protocol. HTTPS is based on SSL, which supports the encryption of messages sent via HTTP.

To secure a communication, HTTPS uses key pairs containing one public key and one private key. Data is encrypted with one key and can only be decrypted with the other key of the key pair. This establishes trust and privacy in message transfers.

To authenticate the Talend Runtime, you need to configure its private key in the `org.ops4j.pax.web.cfg`.

Property	Default	Description
`org.ops4j.pax.web.ssl.keystore`		Path to the keystore file. See http://team.ops4j.org/wiki/display/paxweb/SSL+Configuration for details.
`org.ops4j.pax.web.ssl.keystore.type`	JKS	This property specifies the keystore type. By default, the value is JKS.
`org.ops4j.pax.web.ssl.password`		Password used for keystore integrity check.
`org.ops4j.pax.web.ssl.keypassword`		Password used for keystore.

Advanced configuration

For a complete list of all advanced configuration properties, see http://team.ops4j.org/wiki/display/paxweb/Configuration/.

Default configuration

The Talend Runtime is deployed with the following configuration of the `<TalendRuntimePath>/container/etc/org.ops4j.pax.web.cfg` file.

```
org.osgi.service.http.port=8040
org.osgi.service.http.port.secure=9001
org.osgi.service.http.secure.enabled=true
org.ops4j.pax.web.ssl.keystore=./etc/keystore.jks
org.ops4j.pax.web.ssl.password=password
org.ops4j.pax.web.ssl.keypassword=password
```

Note: The Certificates deployed with the Talend Runtime by default must not be used for production, but only for demo purposes.

Note: And the key password corresponds to the password generated by the user when he/she generated the key.

Enabling client authentication for SSL

To exchange certificates and allow only "trusted" clients to use the Talend Runtime Container HTTP service, you need to follow the following instructions.

1. Enable the HTTP client auth support in the Karaf-based Talend Runtime Container.

When you install the HTTP feature, the container leverages Pax-Web to provide HTTP OSGi service:

```
karaf@trun> feature:install http
```

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

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

org.osgi.service.http.port.secure=9001
org.osgi.service.http.secure.enabled=true
org.ops4j.pax.web.ssl.keystore=./etc/keystores/keystore.jks
org.ops4j.pax.web.ssl.password=password
org.ops4j.pax.web.ssl.keypassword=password
#org.ops4j.pax.web.ssl.clientauthwanted=false
org.ops4j.pax.web.ssl.clientauthneeded=true
```

The `clientauthwanted` and `clientauthneeded` properties are valid for Karaf 2.2.x which uses Pax Web 1.0.x. For more information about the version of Karaf your Talend Runtime Container is based on, see the Talend Installation Guide or the Release Notes.

Thanks to the `clientauthneeded` property, the client is "forced" to be trusted.

Create the trusted client certificate

About this task

You are going to use a keytool (provided with the JDK) to manipulate the keys and certificates.

Procedure

1. Create two key pairs:
   - one for the server side (use for SSL),
   - one as an example of the client side (use for "trust", should be performed for each client, on the client side).

```
mkdir -p etc/keystores
cd etc/keystores
keytool -genkey -keyalg RSA -validity 365 -alias serverkey -keypass password -
storepass password -keystore keystore.jks
keytool -genkey -keyalg RSA -validity 365 -alias clientkey -keypass password -
storepass password -keystore client.jks
```

   These key are self-signed. In a production system, you should use a Certificate Authority (CA).

2. Export the client certificate to be imported in the server keystore:

```
keytool -export -rfc -keystore clientKeystore.jks -storepass password -alias
 clientkey -file client.cer
keytool -import -trustcacerts -keystore keystore.jdk -storepass password -alias
 clientkey -file client.cer
```

3. Check that the client certificate is trusted in our keystore:

```
keytool -list -v -keystore keystore.jks
...
Alias name: clientkey
Creation date: Dec 12, 2012
Entry type: trustedCertEntry
...
```

4. You can now remove the `client.cer` certificate.

Start the container and test with the WebConsole

Procedure

1. Start the Talend Runtime Container:
   - `bin/trun` for Linux
   - `bin/trun.bat` for Windows
2. Install the WebConsole feature:

   ```
   karaf@trun> feature:install webconsole
   ```

   If you try to access to the WebConsole (using a simple browser) using `https://localhost:9001/system/console`, you get the following message:

   ```
   An error occurred during a connection to localhost:9001.

   SSL peer cannot verify your certificate.

   (Error code: ssl_error_bad_cert_alert)
   ```

   Which is normal as the browser does not have any trusted certificate.
3. Add the client certificate in the browser.

   Firefox supports the import of PKCS12 keystore. So, you are going to "transform" the JKS keystore into a PKCS12 keystore:

   ```
   keytool -importkeystore -srckeystore clientKeystore.jks -srcstoretype JKS -
   destkeystore client.pfx -deststoretype PKCS12
   Enter destination keystore password:
   Re-enter new password:
   Enter source keystore password:
   Entry for alias clientkey successfully imported.
   Import command completed: 1 entries successfully imported, 0 entries failed or
    cancelled
   ```

   Now, you can import the client certificate in Firefox. To do so, in the **Tools** menu, click the **Options** entry, and click on the **Advanced** tab.

   You can go in **Certificates** tab and click on **View Certificates** button.

   In the **Your Certificates** tab, you can click on the **Import...** button and choose the `client.pfx` keystore file.
4. If you try to access `https://localhost:9001/system/console` again, you will have access as a trusted client and use it.

Configuring jetty for SSL

To turn off having pax-web to directly create the connector, change the `etc/org.ops4j.pax.web.cfg` file as follows:

```
#org.osgi.service.http.port.secure=9001
#org.osgi.service.http.secure.enabled=true
....
org.ops4j.pax.web.config.file=${karaf.base}/etc/jetty.xml
```

In `etc/jetty.xml`, remove the connector already defined there ("`org.eclipse.jetty.server.nio.BlockingChannelConnector`") and replace it with the following one:

```
<Call name="addConnector">
    <Arg>
        <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
            <Set name="port">9001</Set>
            <Set name="maxIdleTime">30000</Set>
            <Set name="keystore">./etc/keystores/keystore.jks</Set>
            <Set name="password">password</Set>
            <Set name="keyPassword">password</Set>
        </New>
    </Arg>
</Call>
```

Those settings puts the connector on port 9001 to use the **SslSelectChannelConnector** which provides working continuation support.

Client HTTP Configuration

This section discusses configuring HTTPS using the OSGi Configuration Admin Service.

For additional information, see CXF SSL configuration in http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html, which has some sample configurations supported by CXF.

OSGi configuration files

If a Web service is deployed in Talend Runtime as an OSGi bundle, it is now possible to configure SSL via the OSGi Configuration Admin Service. This may be done using the following files (located at `<TalendRuntimePath>/container/etc/`):

- a generic configuration `org.apache.cxf.http.conduits-common.cfg`, used by all HTTPS endpoints in the container,
- endpoint-specific files: `org.apache.cxf.http.conduits-<endpoint_name>.cfg`,
- there may also be additional endpoint-specific configuration files.

Note that instead of the tree-structured XML configurations, these configurations are flat property files. The properties are named after their XML equivalents and are described in detail in HTTP Conduit OSGi Configuration Parameters on page 49.

The non-OSGi style of SSL configuration is supported and if present will take precedence over the OSGi configuration.

HTTP Conduit OSGi Configuration Parameters

The configuration files described in this section are located in `<TalendRuntimePath>/container/etc/ org.apache.cxf.http.conduits-<endpoint_name>.cfg` in the Talend Runtime.

As an example of the syntax involved, here are the contents of the general `org.apache.cxf.http.conduits-common.cfg` configuration file:

```
url = https://localhost.*
tlsClientParameters.disableCNCheck = true
tlsClientParameters.trustManagers.keyStore.type = JKS
tlsClientParameters.trustManagers.keyStore.password = password
tlsClientParameters.trustManagers.keyStore.file = ./etc/keystore.jks
tlsClientParameters.keyManagers.keyStore.type = JKS
tlsClientParameters.keyManagers.keyStore.password = password
tlsClientParameters.keyManagers.keyStore.file = ./etc/keystore.jks
tlsClientParameters.cipherSuitesFilter.include = *_EXPORT_.*,.*_EXPORT\
1024_.*,.*_WITH_DES_.*,.*_WITH_AES_.*,.*_WITH_NULL_.*,.*_DH_anon_.*
```

The `url` parameter

The `url` parameter is one of the main parameters. In the configuration files, the `url` parameter defines the list of matching client endpoints for which the contained configuration parameters are applied. (The client endpoint address is retrieved using `HTTPConduit.getAddress()`).

`url` may be a full endpoint address or may be a regular expression containing wild cards - for example:

- `".*"` matches all endpoints,
- `https.*` matches all client addresses starting with "https".
- `https://localhost.*` matches all client addresses starting with "https" only at localhost.

Note the `org.apache.cxf.http.conduits-common.cfg` file above restricts access to local servers, as the configuration points to a keystore with certificates suitable for the Talend samples. This keystore does not contain the required root certificates from the public certification authorities for public servers, such as Salesforce servers, which are in the standard Java keystore.

All parameters contained in all matching configuration files are collected:

1. in the order defined by the `order` parameter (see the table in the `order` parameter),
2. then by an exact match,
3. then by a configuration with a matching conduit bean name.

If a parameter is defined in multiple matching configuration files, then the last parameter definition found is the one that is used.

The `order` parameter

This parameter defines the order in which the parameters in configuration files are applied. Each file has a unique value of `order`. For example:

`abc.cfg`:

```
order = 1
url = .*
client.ReceiveTimeout = 60000
```

`xyz.cfg`:

```
order = 2
url = .*busy.*
client.ReceiveTimeout = 120000
```

If the endpoint address contains "busy", then both config files match as applicable, according to the rules in the `url` parameter.

In this case, `client.ReceiveTimeout` will have the longer timeout value 120000 because the `order` parameters stipulate that `xyz.cfg` is applied after `abc.cfg`.

Configuration properties

In this table, we look at the complete list of possible properties:

Property	Default	Description
url		The endpoint URL - either defined as exact string or as regular expression pattern (see The url parameter on page 50)
order	50	Defines the order in which parameters are applied (see The order parameter on page 50).
name		If name is defined and is equal to the conduit bean name, `HTTPConduit.getBeanName()`, the parameter definitions have highest priority, overwriting and extending other matching configurations.
tlsClientParameters.secureSocketProtocol	TLS	Protocol Name. Most common examples are "SSL", "TLS" or "TLSv1".
tlsClientParameters.sslCacheTimeout	JDK default	Sets the SSL Session Cache timeout value for client sessions handled by CXF.
tlsClientParameters.jsseProvider		JSSE provider name.
tlsClientParameters.disableCNCheck	false	Indicates whether that the hostname given in the HTTPS URL will be checked against the service's Common Name (CN) given in its certificate during SOAP client requests - it fails if there is a mismatch. If set to true (not recommended for production use), such checks will be bypassed. That will allow you, for example, to use a URL such as localhost during development.
tlsClientParameters.useHttpsURLConnectionDefaultHostname- Verifier	false	This attribute specifies if HttpsURLConnection. getDefaultHostnameVerifier() should be used to create HTTPS connections. If 'true', the 'disableCNCheck' configuration parameter is ignored.
tlsClientParameters.useHttpsURLConnectionDefaultSsl- SocketFactory	false	Specifies if HttpsURLConnection. getDefaultSSLSocketFactory() should be used to create HTTPS connections. If 'true', 'jsseProvider', 'secureSocketProtocol', 'trustManagers', 'keyManagers', 'secureRandom', 'cipherSuites' and 'cipherSuitesFilter' configuration parameters are ignored.
tlsClientParameters.certConstraints.SubjectDNConstraints.combinator		SubjectDN certificate constraints specification as combinator.
tlsClientParameters.certConstraints.SubjectDNConstraints.RegularExpression		SubjectDN certificate constraints specification as regular expression.

Property	Default	Description
tlsClientParameters.certConstraints.Issuer DNConstraints.combinator		IssuerDN certificate constraints specification as combinator.
tlsClientParameters.certConstraints.Issuer DNConstraints.RegularExpression		IssuerDN certificate constraints specification as regular expression.
tlsClientParameters.secureRandomParameters .algorithm	JVM default	algorithm parameter of the SecureRandom specification.
tlsClientParameters.secureRandomParameters .provider	JVM default	provider parameter of the SecureRandom specification.
tlsClientParameters.cipherSuitesFilter.include		filters the supported CipherSuites, list of CipherSuites that will be supported and used if available.
tlsClientParameters.cipherSuitesFilter.exclude		filters the supported CipherSuites, list of CipherSuites that will be excluded.
tlsClientParameters.cipherSuites	default sslContext cipher suites	CipherSuites that will be supported.
tlsClientParameters.trustManagers.provider	JVM default	Provider of the trust manager.
tlsClientParameters.trustManagers.factoryAlgorithm	JVM default	factory algorithm of the trust manager.
tlsClientParameters.trustManagers.keyStore.type	JVM default	Keystore type of the trust manager.
tlsClientParameters.trustManagers.keyStore .password	JVM default	Keystore password of the trust manager.
tlsClientParameters.trustManagers.keyStore.provider	JVM default	Keystore provider of the trust manager.
tlsClientParameters.trustManagers.keyStore.url	JVM default	Trust Managers URL to hold X509 certificates.
tlsClientParameters.trustManagers.keyStore.file	JVM default	Trust Managers file to hold X509 certificates.
tlsClientParameters.trustManagers.keyStore.resource	JVM default	Trust Managers resource to hold X509 certificates.
tlsClientParameters.keyManagers.provider	JVM default	Provider of the key manager.
tlsClientParameters.keyManagers.factoryAlgorithm	JVM default	factory algorithm of the key manager.
tlsClientParameters.keyManagers.keyPassword	JVM default	Key password of the key manager.

Property	Default	Description
tlsClientParameters.keyManagers.keyStore.type	JVM default	Keystore type of the key manager.
tlsClientParameters.keyManagers.keyStore.password	JVM default	Keystore password of the key manager.
tlsClientParameters.keyManagers.keyStore.provider	JVM default	Keystore provider of the key manager.
tlsClientParameters.keyManagers.keyStore.url	JVM default	Key managers URL to hold X509 certificates.
tlsClientParameters.keyManagers.keyStore.file	JVM default	Key managers file to hold X509 certificates.
tlsClientParameters.keyManagers.keyStore.resource	JVM default	Key managers resource to hold X509 certificates.
authorization.UserName		Specifies the UserName parameter for configuring the basic authentication method that the endpoint uses preemptively.
authorization.Password		Specifies the Password parameter for configuring the basic authentication method that the endpoint uses preemptively.
authorization.Authorization		Corresponds to the authentication specified in the SPNEGO/Kerberos login.conf.
authorization.AuthorizationType		Authorization type: "Basic", "Digest" or "Negotiation"
proxyAuthorization.UserName		Specifies the UserName parameter for configuring basic authentication against outgoing HTTP proxy servers.
proxyAuthorization.Password		Specifies the Password parameter for configuring basic authentication against outgoing HTTP proxy servers.
proxyAuthorization.Authorization		Proxy authorization type: "Basic", "Digest" or "Negotiation"
proxyAuthorization.AuthorizationType		Corresponds to the proxy authentication specified in the SPNEGO/Kerberos login.conf.
client.ConnectionTimeout	30000	Specifies the amount of time, in milliseconds, that the client will attempt to establish a connection before it times out. 0 specifies that the client will continue to attempt to open a connection indefinitely.
client.ReceiveTimeout	60000	Specifies the amount of time, in milliseconds, that the client will wait for a response before it times out. 0 specifies that the client will wait indefinitely.
client.AutoRedirect	false	Specifies if the client will automatically follow a server issued redirection. The default is false.

Property	Default	Description
client.MaxRetransmits	-1	Specifies the maximum number of times a client will retransmit a request to satisfy a redirect. The default of -1 specifies that unlimited retransmissions are allowed.
client.AllowChunking	true	Specifies whether the client will send requests using chunking. The default is true which specifies that the client will use chunking when sending requests. Chunking cannot be used if either • http-conf:basicAuthSupplier is configured to provide credentials preemptively or • AutoRedirect is set to true. In both cases the value of `AllowChunking` is ignored and chunking is disallowed. See note about chunking below.
client.ChunkingThreshold	4000	Specifies the threshold at which CXF will switch from non-chunking to chunking. By default, messages less than 4K are buffered and sent non-chunked. Once this threshold is reached, the message is chunked.
client.Connection	Keep-Alive	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • Keep-Alive specifies that the client wants to keep its connection open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • close specifies that the connection to the server is closed after each request/response sequence.
client.DecoupledEndpoint		Specifies the URL of a decoupled endpoint for the receipt of responses over a separate server-client connection. Warning: You must configure both the client and server to use WS-Addressing for the decoupled endpoint to work.
client.ProxyServer		Specifies the URL of the proxy server through which requests are routed.
client.ProxyServerPort		Specifies the port number of the proxy server through which requests are routed.
client.ProxyServerType	HTTP	Specifies the type of proxy server used to route requests. Valid values are: HTTP (default), SOCKS
client.NonProxyHosts		a (possibly empty) list of hosts which should be connected directly and not through the proxy server; it may contain wild card expressions.

Chunking

There are a number of parameters related to chunking, so we discuss this here in more detail. There are two ways of putting the body of a message into an HTTP stream:

1. Standard scheme: this is used by most browsers. It consists in specifying a Content-Length header in the HTTP headers. This allows the receiver to know how much data is coming and when to stop reading.

   The problem with this approach is that the length needs to be pre-determined. The data cannot be streamed as generated as the length needs to be calculated upfront.

   Thus, if chunking is turned off, we need to buffer the data in a byte buffer (or temp file if it is too large) so that the Content-Length can be calculated.
2. Chunked scheme: with this mode, the data is sent to the receiver in chunks. Each chunk is preceded by a hexadecimal chunk size. When a chunk size is 0, the receiver knows that all the data has been received.

   This mode allows better streaming, as we just need to buffer a small amount, up to 8K by default. When the buffer fills, the chunk is written out.

Some parameters in Configuration properties on page 51 allow you to specify the details of the chunking.

In general, the Chunked scheme will perform better as the streaming can take place directly. However, there are some problems with chunking:

- Many proxy servers don't understand it, especially older proxy servers. Many proxy servers want the Content-Length up front so they can allocate a buffer to store the request before passing it onto the real server.
- Some of the older Web Services stacks also have problems with Chunking - specifically, older versions of .NET.

Note: If you are getting strange errors (generally not SOAP faults, but other HTTP-type errors) when trying to interact with a service, try turning off chunking to see if that helps.

Logging System

Karaf provides a powerful logging system based on OPS4j Pax Logging. In addition to being a standard OSGi Log service, it supports the following APIs:

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

Configuration

Configuration file

The configuration of the logging system uses a standard Log4j configuration file at the following location: `<RuntimeContainerPath>/etc/org.ops4j.pax.logging.cfg`.

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

Configuring the appenders

The default logging configuration defines the following appenders:

- the console appender is disabled by default. If you plan to run Karaf in server mode only (i.e. with the locale console disabled), you can turn on this appender on by adding it to the list of configured appenders using the `log4j2.rootLogger.appenderRef` property.
- the `out` appender is the one enabled by default. It logs events to a number of rotating log files of a fixed size. You can easily change the parameters to control the number of files using `appender.rolling.strategy.max` and their size `appender.rolling.policies.size.size`.

Changing the log levels

The default logging configuration sets the logging levels so that the log file will provide enough information to monitor the behavior of the runtime and provide clues about what caused a problem. However, the default configuration will not provide enough information to debug most problems. The most useful logger to change when trying to debug an issue with Karaf is the root logger. You will want to set its logging level to DEBUG in the `org.ops4j.pax.logging.cfg` file.

```
log4j2.rootLogger.level=DEBUG
...
```

Console Log Commands

The log scope comes with several commands -- see Log Scope on page 13 for a full list.

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

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

Note that the log levels set using the log:set commands are not persistent and will be lost upon restart. To configure those in a persistent way, you should edit the configuration file mentioned above

using the config commands or directly using a text editor of your choice. The log commands have a separate configuration file: `<RuntimeContainerPath>/etc/org.apache.karaf.log.cfg`.

For more information about the logging system, see the site http://karaf.apache.org/manual/latest/#_log.

Deployer

The Karaf deployer is used for deploying bundles or groups of bundles (called features) into the Karaf container. The following diagram describes the architecture of the deployer.

Features deployer

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

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

For more informations about features, see Provisioning on page 62.

Spring deployer

Karaf includes a deployer that is able to deploy plain blueprint or spring-dm configuration files. The deployer will transform on the fly any spring configuration file dropped into the deploy folder into a valid OSGi bundle. The generated OSGi manifest will contain the following headers:

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

The name and version of the file are extracted using a heuristic that will match common patterns. For example `my-config-1.0.1.xml` will lead to name = my-config and version = 1.0.1. The default imported packages are extracted from the Spring file definition and includes all classes referenced directly. If you need to customize the generated manifest, you can do so by including an xml element in your Spring configuration:

```
<spring:beans ...>
   <manifest>
      Require-Bundle= my-bundle
   </manifest>
   ...
</spring:beans>
```

Wrap deployer

The wrap deployer allows you to hot deploy non-OSGi jar files ("classical" jar files) from the deploy folder. It's a standard deployer (you don't need to install additional Karaf features):

```
karaf@trun()> la|grep -i wrap
 35 | Active   |  24 | 4.0.0.SNAPSHOT        | Apache Karaf :: Deployer :: Wrap Non
 OSGi Jar
 92 | Active   |   5 | 2.4.1                 | OPS4J Pax Url - wrap:
```

Karaf wrap deployer looks for jar files in the deploy folder. The jar files is considered as non-OSGi if the MANIFEST doesn't contain the Bundle-SymbolicName and Bundle-Version attributes, or if there is no MANIFEST at all. The non-OSGi jar file is transformed into an OSGi bundle. The deployer tries to populate the Bundle-SymbolicName and Bundle-Version extracted from the jar file path. For example, if you simply copy commons-lang-2.3.jar (which is not an OSGi bundle) into the deploy folder, you will see:

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

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

```
karaf@trun> bundle:headers 41

commons-lang (41)
-----------------
Specification-Title = Commons Lang
Tool = Bnd-0.0.357
Specification-Version = 2.3
Specification-Vendor = Apache Software Foundation
Implementation-Version = 2.3
Generated-By-Ops4j-Pax-From = wrap:file:/home/onofreje/workspace/karaf/
    assembly/target/apache-karaf-2.99.99-SNAPSHOT/deploy/commons-lang-2.3
    .jar$ Bundle-SymbolicName=commons-lang&Bundle-Version=2.3
Implementation-Vendor-Id = org.apache
Created-By = 1.6.0_21 (Sun Microsystems Inc.)
Implementation-Title = Commons Lang
Manifest-Version = 1.0
Bnd-LastModified = 1297248243231
X-Compile-Target-JDK = 1.1
Originally-Created-By = 1.3.1_09-85 ("Apple Computer, Inc.")
Ant-Version = Apache Ant 1.6.5
Package = org.apache.commons.lang
X-Compile-Source-JDK = 1.3
Extension-Name = commons-lang
Implementation-Vendor = Apache Software Foundation

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

Import-Package =
        org.apache.commons.lang;resolution:=optional,
        org.apache.commons.lang.builder;resolution:=optional,
        org.apache.commons.lang.enum;resolution:=optional,
        org.apache.commons.lang.enums;resolution:=optional,
        org.apache.commons.lang.exception;resolution:=optional,
```

```
        org.apache.commons.lang.math;resolution:=optional,
        org.apache.commons.lang.mutable;resolution:=optional,
        org.apache.commons.lang.text;resolution:=optional,
        org.apache.commons.lang.time;resolution:=optional
Export-Package =
        org.apache.commons.lang;uses:="org.apache.commons.lang.builder,
           org.apache.commons.lang.math,org.apache.commons.lang.exc
eption",
        org.apache.commons.lang.builder;
             uses:="org.apache.commons.lang.math,org.apache.commons.l
ang",
        org.apache.commons.lang.enum;uses:=org.apache.commons.lang,
        org.apache.commons.lang.enums;uses:=org.apache.commons.lang,
        org.apache.commons.lang.exception;uses:=org.apache.commons.lang,
        org.apache.commons.lang.math;uses:=org.apache.commons.lang,
        org.apache.commons.lang.mutable;uses:="org.apache.commons.lang,
             org.apache.commons.lang.math",
        org.apache.commons.lang.text;uses:=org.apache.commons.lang,
        org.apache.commons.lang.time;uses:=org.apache.commons.lang
```

You may set the manifest headers by specifying them as URL parameters. The token '$' is used to separate the hierarchy path and query parts of the URL, with standard URL syntax for query parameters of key=value pairs joined by '&'. On the command line you must use single quotes around the URL to prevent the shell from attempting variable expansion, and if present, you must escape exclamation marks with a backslash.

```
install -s 'wrap:mvn:jboss/jbossall-client/4.2.3.GA/   \\
    $Bundle-SymbolicName=jbossall-client&Bundle-Version=4.2.3.GA&   \\
    Export-Package=org.jboss.remoting;version="4.2.3.GA",\!*'
```

If defined in a features.xml file, you'll need 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   \\
    &Bundle-Version=4.3.2.GA   \\
    &Export-Package=org.jboss.remoting;version="4.3.2.GA",!*
</bundle>
```

War deployer

See Web Applications on page 68 for information on web application (war) deployment.

Servlet Context

A servlet context defines a set of methods which allows a servlet to communicate with its servlet container. Karaf and CXF provide servlet context custom configuration, for building services.

For example, you can deploy into a servlet container, using a servlet transport, `CXFServlet`. The following section explains servlet context configuration in more detail.

Please read the Servlet Transport on the Apache CXF website for additional information about this servlet context.

Servlet context configuration

First we look at the configuration files. The configuration of the servlet context for the OSGi HTTP Service is specified in a file at the following location: `<RuntimeContainerPath>/etc/org.apache.cxf.osgi.cfg`

The configuration of the port for the OSGi HTTP Service is specified in: `<RuntimeContainerPath>/etc/org.ops4j.pax.web.cfg`

You can edit these files at runtime and any change will be reloaded and be effective immediately.

Configuring the context

When editing the files, the `org.apache.cxf.osgi.cfg` file specified prefix for the services is:

```
org.apache.cxf.servlet.context=/services
```

The `org.ops4j.pax.web.cfg` file specified port for the services is:

```
org.osgi.service.http.port=8040
```

Relative endpoint address

The CXFServlet uses a relative address for the endpoint rather than a full http address. For example, given an implementation class called `GreeterImpl` with endpoints `greeter` and `greeterRest`, the relative endpoint addresses would be configured as:

```
<jaxws:endpoint id="greeter"
   implementor="org.apache.hello_soap_http.GreeterImpl"
   address="/Greeter1"/>

<jaxrs:server id="greeterRest"
   serviceClass="org.apache.hello_soap_http.GreeterImpl"
   address="/GreeterRest"/>
```

The cumulative result of these changes is that the endpoint address for the servlet will be: `http://{server}:8040/services/Greeter1` and `http://{server}:8040/services/GreeterRest`

Provisioning

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

Example: Deploying a sample feature

In the rest of this chapter, we describe in detail the process involved in provisioning. But as a quick demonstration, we'll run a sample Apache Camel feature already present in the Talend ESB distribution. In the console, run the following commands:

```
feature:repo-add mvn:org.apache.camel/camel-example-osgi/2.10.0/xml/features
feature:install camel-example-osgi
```

The example installed uses Camel to start a timer every 2 seconds and output a message on the console. These `feature:repo-add` and `feature:install` commands download the Camel features descriptor and install this example. The output is as follows:

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

Stopping and uninstalling the sample application

To stop this demo, run the following command:

```
feature:uninstall camel-example-osgi
```

Repositories

So, first we look at feature repositories. The features are described in a features XML descriptor, a "features repository". This features XML has a schema, so for more information on this schema, see **Features XML Schema** in the **Provisioning** section of the lastest version of the *Karaf User Guide* (http://karaf.apache.org/). We recommend using this XML schema. It will allow Karaf to validate your repository before parsing. You may also verify your descriptor before adding it to Karaf by simply validation, even from IDE level. Here is an example of such a repository:

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

A repository includes a list of feature elements, each one representing an application that can be installed. The feature is identified by its name which must be unique amongst all the

repositories used and consists of a set of bundles that need to be installed along with some optional dependencies on other features and some optional configurations for the Configuration Admin OSGi service.

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

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

Be careful when you define them as there is a risk of 'cycling' dependencies. Note: By default, all the features defined in a repository are not installed at the launch of Apache Karaf (see Service configuration on page 67 for more information).

Bundles

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

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

Doing this will make sure the above bundle is installed while installing the feature. However, Karaf provides several URL handlers, in addition to the usual ones (file, http, etc...). One of these is the Maven URL handler, which allow reusing Maven repositories to point to the bundles.

Maven URL Handler

The equivalent of the above bundle would be:

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

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

The org.ops4j.pax.url.mvn bundle resolves mvn URLs. This flexible tool can be configured through the configuration service. For example, to find the current repositories type: `karaf@trun:/> config:list` and the following will display:

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

The repositories checked are controlled by these configuration properties. For example, org.ops4j.pax.url.mvn.repositories is a comma separated list of repository URLs specifying those remote repositories to be checked. So, to replace the defaults with a new repository at http://www.example.org/repo on the local machine:

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

By default, snapshots are disabled. To enable an URL for snapshots append @snapshots. For example: http://www.example.org/repo@snapshots. Repositories on the local are supported through file:/ URLs.

Bundle start-level

By default, the bundles deployed through the feature mechanism will have a start-level equals to the value defined in the configuration file config.properties with the variable karaf.startlevel.bundle=60. This value can be changed using the xml attribute start-level.

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

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

Bundle stop/start

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

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

Bundle dependency

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

Dependent Features

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

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

The effect of such a dependency is to automatically install the required nmr feature when the jbi feature will be installed. A version range can be specified on the feature dependency:

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

In such a case, if no matching feature is already installed, the feature with the highest version available in the range will be installed. If a single version is specified, this version will be chosen. If nothing is specified, the highest available will be installed.

Configurations

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

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

The name attribute of the configuration element will be used as the ManagedService PID for the configuration set in the Configuration Admin service. When using a ManagedServiceFactory, the name attribute is servicePid-_aliasId_, where servicePid is the PID of the ManagedServiceFactory and aliasId is a label used to uniquely identify a particular service (an alias to the factory generated service PID). Deploying such a configuration has the same effect than dropping a file named `com.foo.bar.cfg` into the `etc` folder.

The content of the configuration element is set of properties parsed using the standard java property mechanism. Such configuration as usually used with Spring-DM or Blueprint support for the Configuration Admin service, as in the following example, but using plain OSGi APIs will of course work the same way:

```
<bean ...>
   <property name="propertyName" value="${myProperty}" />
</bean>
<osgix:cm-properties id="cmProps" persistent-id="com.foo.bar">
   <prop key="myProperty">myValue</prop>
</osgix:cm-properties>
<ctx:property-placeholder properties-ref="cmProps" />
```

There may also be cases where you want to make the properties from multiple configuration files available to your bundle context. This is something you may want to do if you have a multi-bundle application where there are application properties used by multiple bundles, and each bundle has

its own specific properties. In that case, <ctx:property-placeholder> won't work as it was designed to make only one configuration file available to a bundle context. To make more than one configuration file available to your bundle-context you would do something like this:

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

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

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

Additional configuration files

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

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

Feature resolver

The resolver attribute on a feature can be set to force the use of a given resolver instead of the default resolution process. A resolver will be used to obtain the list of bundles to actually install for a given feature.

The default resolver will simply return the list of bundles provided in the feature description. The obr resolver can be installed and used instead of the standard one. In that case, the resolver will use the OBR service to determine the list of bundles to install (bundles flagged as dependency will only be used as possible candidates to solve various constraints).

Commands

Repository management

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

- `feature:repo-add`: Add a list of repository URLs to the features service
- `feature:repo-remove`: Remove a list of repository URLs from the features service

- `feature:repo-list`: Display the repository URLs currently associated with the features service.
- `feature:repo-refresh`: Reload the repositories to obtain a fresh list of features

Karaf maintains a persistent list of these repositories so that if you add one URL and restart Karaf, the features will still be available. The `repo-refresh` command is mostly used when developing features descriptors: when changing the descriptor, it can be handy to reload it in the Kernel without having to restart it or to remove then add again this URL.

Features management

Common feature: scope commands used in features management include `feature:install`, `feature:uninstall`, and `feature:list`. See Feature Scope on page 11 for more information on these commands.

Examples

To install features using mvn handler:

```
feature:repo-add mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.0.0-m2/xml/featu
res
feature:install nmr
```

To use a file handler to deploy a features file (note the path is relative to the Apache Karaf installation directory):

```
feature:repo-add file:base/features/features.xml
```

To deploy bundles from file system without using Maven: As we can use file:// as protocol handler to deploy bundles, you can use the following syntax to deploy bundles when they are located in a directory which is not available using Maven:

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

Note the path again is relative to Apache Karaf installation directory.

Service configuration

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

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

This configuration file is of interest if you plan to distribute a Apache Karaf distribution which includes pre-installed features.

Web Applications

Karaf provides an ability to deploying WAR-based web applications within the Jetty server contained in the Karaf instance.

Installing WAR support

About this task

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

Procedure

1. List the available features:

```
karaf@trun> feature:list
State         Name
...
[uninstalled] [2.2.9] obr         karaf-2.2.9
[uninstalled] [2.2.9] config      karaf-2.2.9
[uninstalled] [2.2.9] http        karaf-2.2.9
[uninstalled] [2.2.9] war         karaf-2.2.9
[uninstalled] [2.2.9] webconsole karaf-2.2.9
[installed  ] [2.2.9] ssh         karaf-2.2.9
...
```

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

```
karaf@trun> feature:install war
```

   Note: you can use the -v or --verbose switch to see exactly what Karaf does, and this will show the war, http and jetty features being installed, along with dependent bundles.

3. Verify the features were installed:

```
karaf@trun> feature:list
State         Name
...
[installed ] [2.2.9] http karaf-2.2.9
[installed ] [2.2.9] war  karaf-2.2.9
...
```

4. Verify the installed bundles were started

```
karaf@trun> list
START LEVEL 100
ID   State         Level  Name
...
[ 32] [Active ] [ ] [ 60] geronimo-servlet_2.5_spec (1.1.2)
[ 33] [Active ] [ ] [ 60] Apache ServiceMix :: Bundles ::
      jetty (6.1.22.2)
[ 34] [Active ] [ ] [ 60] OPS4J Pax Web - API (1.0.0)
[ 35] [Active ] [ ] [ 60] OPS4J Pax Web - Service SPI (1.0.0)
[ 36] [Active ] [ ] [ 60] OPS4J Pax Web - Runtime (1.0.0)
[ 37] [Active ] [ ] [ 60] OPS4J Pax Web - Jetty (1.0.0)
[ 38] [Active ] [ ] [ 60] OPS4J Pax Web - Jsp Support (1.0.0)
[ 39] [Active ] [ ] [ 60] OPS4J Pax Web - Extender - WAR (1.0.0)
[ 40] [Active ] [ ] [ 60] OPS4J Pax Web - Extender - Whiteboard
```

```
(1.0.0)
[ 42] [Active ] [ ] [ 60] OPS4J Pax Web - FileInstall Deployer
(1.0.0)
[ 41] [Active ] [ ] [ 60] OPS4J Pax Url - war:, war-i: (1.2.4)
...
```

Note: the version numbers of the packages may be different in your installation.

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

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

Deploying a WAR to the installed web feature

About this task

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

Procedure

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

Results

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

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

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

Monitoring and Administration using JMX

Apache Karaf provides a large set of MBeans that allow you to fully monitor and administrate Karaf using any JMX client (for example, JConsole provided in the Oracle or IBM JDK). They provide more or less the same actions that you can do using the Karaf shell commands.

The list of MBeans available:

For MBean registration, the Karaf container exposes a MBeanServer as an OSGI service. This server is created when Karaf boots and can be used to register your own MBeans. The process is really simple as it only requires that the bean/pojo of the project extend the javax.management.StandardMBean class and implement an interface where the name of the class must contain the extension MBean. Below shows an example based on Karaf WebMBean:

Installing the Talend Runtime Container as a service

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

To install Talend Runtime Container as a service, you first have to install the wrapper, which is an optional feature, and you can then install the service.

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

Supported platforms

The following platforms are supported by the Wrapper:

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

Installing the wrapper

First, to install the wrapper, simply:

1. Browse to the `bin` folder of the Talend Runtime Container directory, then launch:
   - `trun.bat` in Administrator mode on Windows
   - `trun` as root user on Linux
2. To install the wrapper feature, simply type:
   - `karaf@trun> feature:install wrapper` on Windows.
   - `trun@root> feature:install wrapper` on Linux.

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

```
trun@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.
   --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 null)
```

3. To set up the installation of the service, type in the following command:
   - `karaf@trun> wrapper:install` on Windows.
   - `trun@root> wrapper:install` on Linux.

   Warning:

   In Windows you may get the following message when attempting to install the service:

   ```
   wrapper  | OpenSCManager failed - Access is denied. (0x5)
   ```

   In this case launch the Command Prompt in Administrator mode. Right-click **Command Prompt** in the **Start** menu and select **Run as administrator**.

For instance, to register Talend Runtime Container as a service (depending on the running OS), in automatic start mode, simply type:

- For Windows:

```
karaf@trun> wrapper:install -s AUTO_START -n TALEND-ESB-CONTAINER  -d Talend-ESB-
Container -D "Talend ESB Container Service"
```

- For Linux:

```
trun@root> wrapper:install -s AUTO_START -n TALEND-ESB-CONTAINER  -d Talend-ESB-Con
tainer -D "Talend ESB Container Service"
```

Here is an example of `wrapper:install` command executing on Windows (Note: this also contains instructions on how to remove the service):

```
karaf@trun> wrapper:install -s AUTO_START -n TALEND-ESB-CONTAINER
-d Talend-ESB-Container  -D "Talend ESB Container Service"

Creating file: C:\work\-release\Talend-ESB-V\container\bin\
TALEND-ESB-CONTAINER-wrapper.exe
Creating file: C:\work\-release\Talend-ESB-V\container\etc\
TALEND-ESB-CONTAINER-wrapper.conf
Creating file: C:\work\-release\Talend-ESB-V\container\bin\
TALEND-ESB-CONTAINER-service.bat
Creating file: C:\work\-release\Talend-ESB-V\container\lib\
```

```
wrapper.dll
Creating file: C:\work\-release\Talend-ESB-V\container\lib\
karaf-wrapper.jar
Creating file: C:\work\-release\Talend-ESB-V\container\lib\
karaf-wrapper-main.jar

Setup complete.  You may wish to tweak the JVM properties in the
wrapper configuration file:
        C:\work\-release\Talend-ESB-V\container\etc\TALEND-
ESB-CONTAINER-wrapper.conf
before installing and starting the service.

To install the service, run:
  C:> C:\work\-release\Talend-ESB-V\container\bin\
TALEND-ESB-CONTAINER-service.bat install

Once installed, to start the service run:
  C:> net start "TALEND-ESB-CONTAINER"

Once running, to stop the service run:
  C:> net stop "TALEND-ESB-CONTAINER"

Once stopped, to remove the installed the service run:
  C:> C:\work\-release\Talend-ESB-V\container\bin\
TALEND-ESB-CONTAINER-service.bat remove
```

Here is an example of `wrapper:install` command executing on Linux:

```
trun@root> wrapper:install -s AUTO_START -n TALEND-ESB-CONTAINER  \
        -d Talend-ESB-Container -D "Talend ESB Container Service"
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
bin/KARAF-wrapper
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
bin/KARAF-service
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
etc/KARAF-wrapper.conf
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
lib/libwrapper.so
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
lib/karaf-wrapper.jar
Creating file: /home/onofreje/-release/Talend-ESB-V/container/
lib/karaf-wrapper-main.jar

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

Installing the service

On Windows

Procedure

1. Open a CMD window in Administrator mode.
2. Browse to the `bin` folder of the Talend Runtime installation directory, then type in the following command:

   ```
   TALEND-ESB-CONTAINER-service install
   ```

Here is an example of the installation of Talend Runtime Container as a service on Windows:

```
C:\Builds\Talend-Runtime\bin>TALEND-ESB-CONTAINER-service.bat install

wrapper | Talend ESB Container installed.
```

The Talend Runtime service is created and can be viewed by selecting Control Panel > Administrative Tools > Services in the Start menu of Windows.

3. You can then run the `net start "TALEND-CONTAINER"` and `net stop "TALEND-ESB-CONTAINER"` commands to manage the service.

Results

To remove the service, type in the following command in the command window:

```
TALEND-ESB-CONTAINER-service.bat remove
```

On Linux

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

On Redhat/Fedora/CentOS Systems

To install the service:

```
$ ln -s /home/onofreje/-release/Talend-ESB-V/container/bin/TALEND-ESB-CONTAINER-service
  /etc/init.d/
$ chkconfig TALEND-ESB-CONTAINER-service --add
```

To start the service when the machine is rebooted:

```
$ chkconfig TALEND-ESB-CONTAINER-service on
```

To disable starting the service when the machine is rebooted:

```
$ chkconfig TALEND-ESB-CONTAINER-service off
```

To start the service:

```
$ service TALEND-ESB-CONTAINER-service start
```

To stop the service:

```
$ service TALEND-ESB-CONTAINER-service stop
```

To uninstall the service:

```
$ chkconfig TALEND-ESB-CONTAINER-service --del
$ rm /etc/init.d/TALEND-ESB-CONTAINER-service
```

On Ubuntu/Debian Systems

To install the service:

```
$ ln -s /home/onofreje/-release/Talend-ESB-V/container/bin/TALEND-ESB-CONTAINER-
service /etc/init.d/
```

To start the service when the machine is rebooted:

```
$ update-rc.d TALEND-ESB-CONTAINER-service defaults
```

To disable starting the service when the machine is rebooted:

```
$ update-rc.d -f TALEND-ESB-CONTAINER-service remove
```

To start the service:

```
$ /etc/init.d/TALEND-ESB-CONTAINER-service start
```

To stop the service:

```
$ /etc/init.d/TALEND-ESB-CONTAINER-service stop
```

To uninstall the service:

```
$ rm /etc/init.d/TALEND-ESB-CONTAINER-service
```

Configuration Hints

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

Setting up the Talend ESB Active/Passive configuration

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.

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.

So, Talend ESB, based on Apache Karaf, benefits from those mechanisms.

HA/failover (active/passive)

The Talend ESB 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 Talend ESB 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, and so on).

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 HA/failover mechanism.
- `karaf.lock.class` property contains the class name providing the lock implementation. Here, the filesystem lock is used.
- `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 is not always possible and easy to have a shared filesystem between multiple Talend ESB instances.

Instead of sharing a filesystem, Talend ESB supports the sharing of 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, acquires 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.DefaultJDBCLock
karaf.lock.level=50
karaf.lock.delay=10
karaf.lock.jdbc.url=jdbc:derby://dbserver:1527/sample
karaf.lock.jdbc.driver=org.apache.derby.jdbc.ClientDriver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

- `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.DefaultJDBCLock` is the most generic database lock system implementation. Apache Karaf supports lock system for specific databases. For more information, see the sections: Lock on Oracle on page 78, Lock on Derby on page 78, Lock on MySQL on page 78, Lock on PostgreSQL on page 79, and Lock on Microsoft SQLServer on page 79.
- `karaf.lock.level` property is the container-level locking. For more information, see the sections: Lock on Oracle on page 78, Lock on Derby on page 78, Lock on MySQL on page 78, Lock on PostgreSQL on page 79, and Lock on Microsoft SQLServer on page 79.
- `karaf.lock.delay` property is the interval period (in seconds) to check if the lock has been released or not.
- `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 connect to the database.
- `karaf.lock.jdbc.table` property contains the database table to use for the lock.

The Talend ESB will not start if the JDBC driver is not present in the `lib/ext` folder.

The `sample` database will be created automatically if it does not exist.

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 required a manual restart.

Lock on Oracle

Talend ESB 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.

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

Talend ESB 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

Talend ESB 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

Talend ESB 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

Talend ESB 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=10
karaf.lock.jdbc.url=jdbc:jtds:sqlserver://127.0.0.1;databaseName=sample
karaf.lock.jdbc.driver=net.sourceforge.jtds.jdbc.Driver
karaf.lock.jdbc.user=user
karaf.lock.jdbc.password=password
karaf.lock.jdbc.table=KARAF_LOCK
karaf.lock.jdbc.clustername=karaf
karaf.lock.jdbc.timeout=30
```

The JTDS JDBC driver file has to be copied in the `lib/ext` folder.

Container-level locking

Talend ESB 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 Talend ESB 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 Talend ESB 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.

Level	Behavior
<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.

Using 'hot' standby means that the slave instances are running and bind 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.

Troubleshooting Talend ESB

This chapter describes how to fix problems related to JVM memory allocation.

Memory Allocation Parameters

The Talend ESB start scripts define the JVM memory allocation parameters used for running Talend ESB.

JAVA_MIN_MEM. Determines the start size of the Java heap memory. Higher values may reduce garbage collection and improve performance. Corresponds to the JVM parameter -Xms.

JAVA_MAX_MEM. Determines the maximum size of the Java heap memory. Higher values may reduce garbage collection, improve performance and avoid "Out of memory" exceptions. Corresponds to the JVM parameter -Xmx.

JAVA_PERM_MEM. Determines the start size of permanent generation. This is a separate heap space that is normally not garbage collected. Java classes are loaded in permanent generation. Using Talend ESB many classes are loaded, specially when using security features and deploying multiple services to the same container. Increase if a "PermGen space" exception is thrown. Corresponds to the JVM parameter -XX:PermSize

JAVA_MAX_PERM_MEM. Determines the maximum size of permanent generation. This is a separate heap space that is normally not garbage collected. Increase if a "PermGen space" exception is thrown. Corresponds to the JVM parameter -XX:MaxPermSize

Additionally the following parameters are used as default to enable class unloading from permanent generation which reduces the overall permanent generation space usage: -XX:+CMSClassU nloadingEnabled and -XX:+UseConcMarkSweepGC.

Warning: If you set JAVA_OPTS yourself from outside or inside the `<RuntimeContainerPath>/bin/setenv` script you overwrite all default settings provided by karaf including the memory settings described above. This could be useful if you need additional options. On Linux/Solaris you can set DUMP_JAVA_OPTS=true in setenv to find out how karaf would set JAVA_OPTS so you can copy or adapt these settings to fit your requirements.

On Windows

Adapt the memory allocation parameters in the Karaf start script at the following location: `<RuntimeContainerPath>/bin/setenv.bat`.

In configuring the script variable

```
%Bit_64%
```

you can set different values depending whether your JVM is 64-bit. 64-bit-JVMs need significantly more memory.

Warning:

There is known issue related 32-bit JVM on 64-bit Windows.

If you set JAVA_OPTS yourself you might get errors during initialization of VM. Because of this problem you would not be able to run tesb container or build examples. In this case, we recommend to set following values for memory parameters: JAVA_OPTS=-Xmx512m -XX:MaxPermSize=256m

On Linux/Solaris

Adapt the memory allocation parameters in the Karaf start script at the following location: `<RuntimeContainerPath>/bin/setenv`.

In configuring the script variable

```
$JAVA_SIXTY_FOUR
```

you can set different values depending whether your JVM is 64-bit. 64-bit-JVMs need significantly more memory.

On Solaris, there is an additional flag JAVA_64BIT_SOLARIS, which is set in `<RuntimeContainerPath>/bin/setenv`. The default is JAVA_64BIT_SOLARIS=true. This flag is evaluated only for Solaris and determines whether the JVM operates in 64-bit or 32-bit mode. This is necessary, because in Solaris (only), a JVM capable of 64-bit starts normally in 32-bit mode. Set JAVA_64BIT_SOLARIS=false if you really want to choose 32-bit mode.

Note that in Redhat Linux Enterprise 5/64 Bit, the memory allocation is different depending on whether you use the container through the wrapper utility or not. If you have installed the service with the wrapper feature, you need to tweak the JVM parameters in the wrapper.conf file, see http://wrapper.tanukisoftware.com/doc/english/prop-java-maxmemory.html for details.