# [Karaf Tutorial Part 1 - Installation and First application](http://www.liquid-reality.de/display/liquid/2011/02/15/Karaf+Tutorial+Part+1+-+Installation+and+First+application)

[Skip to end of metadata](http://www.liquid-reality.de/display/liquid/2011/02/15/Karaf+Tutorial+Part+1+-+Installation+and+First+application#page-metadata-end)

* Created by [Christian Schneider](http://www.liquid-reality.de/display/%7Echris), last modified on [Jul 02, 2015](http://www.liquid-reality.de/pages/diffpagesbyversion.action?pageId=5865484&selectedPageVersions=9&selectedPageVersions=10)

[Go to start of metadata](http://www.liquid-reality.de/display/liquid/2011/02/15/Karaf+Tutorial+Part+1+-+Installation+and+First+application#page-metadata-start)

# Getting Started

With this post I am beginning a series of posts about Apache Karaf. So what is Karaf and why should you be interested in it? Karaf is an OSGi container based on Equinox or Felix. The main difference to these fine containers is that it brings excellent management features with it.

Outstanding features of Karaf:

* Extensible Console with Bash like completion features
* ssh console
* deployment of bundles and features from maven repositories
* easy creation of new instances from command line

All together these features make developing server based OSGi applications almost as easy as regular java applications. Deployment and management is on a level that is much better than all applications servers I have seen till now. All this is combined with a small footprint as well of karaf as the resulting applications. In my opinion this allows a light weight development style like JEE 6 together with the flexibility of spring applications.

# Installation and first startup

* Download Karaf 3.0.3 from the [Karaf web site](http://karaf.apache.org/).
* Extract and start with bin/karaf

You should see the welcome screen:

|  |
| --- |
| \_\_ \_\_                  \_\_\_\_         / //\_/\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_/ \_\_/        / ,<  / \_\_ `/ \_\_\_/ \_\_ `/ /\_       / /| |/ /\_/ / /  / /\_/ / \_\_/      /\_/ |\_|\\_\_,\_/\_/   \\_\_,\_/\_/      Apache Karaf (3.0.3)    Hit '<tab>' for a list of available commands  and '[cmd] \--help' for help on a specific command.  Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown Karaf.    karaf@root()> |

# Some handy commands

| **Command** | **Description** |
| --- | --- |
| la | Shows all installed bundles |
| service:list | Shows the active OSGi services. This list is quite long. Here it is quite handy that you can use unix pipes like "ls | grep admin" |
| exports | Shows exported packages and bundles providing them. This helps to find out where a package may come from. |
| feature:list | Shows which features are installed and can be installed. |
| features:installwebconsole | Install features (a list of bundles and other features). Using the above command we install the Karaf webconsole.  features:install webconsole  It can be reached at <http://localhost:8181/system/console> . Log in with karaf/karaf and take some time to see what it has to offer.  features:install webconsole |
| log:tail | Show the log. Use ctrl-c to  go back to Console |
| Ctrl-d | Exit the console. If this is the main console karaf will also be stopped. |

OSGi containers preserve state after restarts

Icon

Please note that Karaf like all osgi containers maintains it´s last state of installed and started bundles. So if something should not work anymore a restart is not sure to help. To really start fresh again stop karaf and delete the data directory.

Check the logs

Icon

Karaf is very silent. To not miss error messages always keep a tail -f data/karaf.log open !!

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

features:install camel-example-osgi

features:uninstall camel-example-osgi

# Tasklist - A small osgi application

Without any useful application Karaf is a nice but useless container. So let´s create our first application. The good news is that creating an OSGi application is quite easy and  
maven can help a lot. The difference to a normal maven project is quite small. To write the application I recommend to use Eclipse 4 with the m2eclipse plugin which is installed by default on current versions.

Get the source code

* Clone the example using git clone [git@github.com](mailto:git@github.com):cschneider/Karaf-Tutorial.git
* or
* download the sample project from <https://github.com/cschneider/Karaf-Tutorial/zipball/master> and extract to a directory

Import into Eclipse

* Start Eclipse
* In Eclipse Package explorer: Import -> Existing maven project -> Browse to the extracted directory into the tasklist sub dir
* Eclipse will show all maven projects it finds
* Click through to import with defaults

Eclipse will now import the projects and wire all dependencies using m2eclipse.

The [tasklist example](https://github.com/cschneider/Karaf-Tutorial/tree/master/tasklist) consists of three projects

| **Module** | **Description** |
| --- | --- |
| tasklist-model | Service interface and Task class |
| tasklist-persistence | Simple persistence implementation that offers a TaskService |
| tasklist-ui | Servlet that displays the tasklist using a TaskService |
| tasklist-features | Features descriptor for the application that makes installing in Karaf very easy |

## Tasklist-persistence

This project contains the domain model and the service implementation. The model is the Task class and a TaskService interface. The persistence implementation TaskServiceImpl manages tasks in a simple HashMap.  
The TaskService is published as an OSGi service using a blueprint context. Blueprint is an OSGi standard for dependency injection and is very similar to a spring context.

|  |
| --- |
| <blueprint xmlns="[http://www.osgi.org/xmlns/blueprint/v1.0.0"](http://www.osgi.org/xmlns/blueprint/v1.0.0)>      <bean id="taskService" class="net.lr.tasklist.persistence.impl.TaskServiceImpl" />      <service ref="taskService" interface="net.lr.tasklist.model.TaskService" />  </blueprint> |

The bean tag creates a single instance of the TaskServiceImpl. The service tag publishes this instance as an OSGi service with the TaskService interface.

The pom.xml is of packaging bundle and the maven-bundle-plugin creates the jar with an OSGi Manifest. By default the plugin imports all packages that are imported in java files or referenced in the blueprint context.  
It also exports all packages that do not contain the string impl or internal. In our case we want the model package to be imported but not the persistence.impl package. As the naming convention is used  
we need no additional configuration.

## Tasklist-ui

The ui project contains a small servlet TaskServlet to display the tasklist and individual tasks. To work with the tasks the servlet needs the TaskService.

To inject the TaskService and to publish the servlet the following blueprint context is used:

|  |
| --- |
| <blueprint xmlns="[http://www.osgi.org/xmlns/blueprint/v1.0.0"](http://www.osgi.org/xmlns/blueprint/v1.0.0)>      <reference id="taskService" availability="mandatory" interface="net.lr.tasklist.model.TaskService" />      <bean id="taskServlet" class="net.lr.tasklist.ui.TaskListServlet">          <property name="taskService" ref="taskService"></property>      </bean>      <service ref="taskServlet" interface="javax.servlet.http.HttpServlet">          <service-properties>              <entry key="alias" value="/tasklist" />          </service-properties>      </service>  </blueprint> |

The reference tag makes blueprint search and eventually wait for a service that implements the TaskService interface and creates a bean "taskService".  
The bean taskServlet instantiates the servlet class and injects the taskService. The service tag publishes the servlet as an OSGi service with the HttpServlet interface and sets a property alias.  
This way of publishing a servlet is not yet standardized but is supported by the pax web whiteboard extender. This extender registers each service with interface HttpServlet with the OSGi http service. It uses the alias  
property to set the path where the servlet is available.

See also: <http://wiki.ops4j.org/display/paxweb/Whiteboard+Extender>

## Tasklist-features

The last project only installs a feature descriptor to the maven repository so we can install it easily in Karaf. The descriptor defines a feature named tasklist and the bundles to be installed from the maven repository.

|  |
| --- |
| <feature name="example-tasklist-persistence" version="${pom.version}">      <bundle>mvn:net.lr.tasklist/tasklist-model/${pom.version}</bundle>      <bundle>mvn:net.lr.tasklist/tasklist-persistence/${pom.version}</bundle>  </feature>    <feature name="example-tasklist-ui" version="${pom.version}">      <feature>http</feature>      <feature>http-whiteboard</feature>      <bundle>mvn:net.lr.tasklist/tasklist-model/${pom.version}</bundle>      <bundle>mvn:net.lr.tasklist/tasklist-ui/${pom.version}</bundle>  </feature> |

A feature can consist of other features that also should be installed and bundles to be installed. The bundles typically use mvn urls. This means they are loaded from the configured maven repositories or your local maven repositiory in ~/.m2/repository.

# Installing the Application in Karaf

|  |
| --- |
| feature:repo-add mvn:net.lr.tasklist/tasklist-features/1.0.0-SNAPSHOT/xml  feature:install example-tasklist-persistence example-tasklist-ui |

Add the features descriptor to Karaf so it is added to the available features, then Install and start the tasklist feature. After this command the tasklist application should run

|  |
| --- |
| list |

Check that all bundles of tasklist are active. If not try to start them and check the log.

|  |
| --- |
| http:list    ID | Servlet         | Servlet-Name   | State       | Alias     | Url  -------------------------------------------------------------------------------  56 | TaskListServlet | ServletModel-2 | Deployed    | /tasklist | [/tasklist/\*] |

Should show the TaskListServlet. By default the example will start at <http://localhost:8181/tasklist> .

You can change the port by creating aa text file in "etc/org.ops4j.pax.web.cfg" with the content "org.osgi.service.http.port=8080". This will tell the HttpService to use the port 8080. Now the tasklist application should be available at <http://localhost:8080/tasklist>

# Summary

In this tutorial we have installed Karaf and learned some commands. Then we created a small OSGi application that shows servlets, OSGi services, blueprint and the whiteboard pattern.

# [Karaf Tutorial Part 2 - Using the Configuration Admin Service](http://www.liquid-reality.de/display/liquid/2011/09/23/Karaf+Tutorial+Part+2+-+Using+the+Configuration+Admin+Service)

[Skip to end of metadata](http://www.liquid-reality.de/display/liquid/2011/09/23/Karaf+Tutorial+Part+2+-+Using+the+Configuration+Admin+Service#page-metadata-end)

* Created by [Christian Schneider](http://www.liquid-reality.de/display/%7Echris), last modified on [Jul 20, 2012](http://www.liquid-reality.de/pages/diffpagesbyversion.action?pageId=6193179&selectedPageVersions=6&selectedPageVersions=7)

[Go to start of metadata](http://www.liquid-reality.de/display/liquid/2011/09/23/Karaf+Tutorial+Part+2+-+Using+the+Configuration+Admin+Service#page-metadata-start)

In the first part of the Karaf Tutorial we learned how to use maven and blueprint to offer and use pojo services and how to use the http service to publish a servlet.

In this second part we concentrate on configuration for our OSGi bundles. Unlike servlet containers OSGi contains a very good specification for configuration: The Config Admin Service from the OSGi enterprise spec. In this tutorial we will cover ussing the Config Admin Service with pure OSGi and blueprint and how to automatically deploy config files with your bundles.

The practical parts of this tutorial can be found on github in <https://github.com/cschneider/Karaf-Tutorial/tree/master/configadmin>

## The Configuration Admin Service spec

We will first get a fast overview of the Configuration Admin Service spec. There two main interfaces for us to use:

* ConfigurationAdmin - Allows to retrieve and change configurations. This service is offered by the Config Admin Service implementation
* ManagedService - Allows to react on configuration changes. You have to implement this and register it as a service to get notified

So basically a configuration in the Config Admin Service is a Dictionary that contains attributes and their values. The Dictionary is identified by a persistent identifier (pid). This is simply a String that should uniquely identify the configuration.

## How to work with configuration?

While you can retrieve a configuration using the ConfigurationAdmin.getConfiguration interface I would not recommend to do so. OSGi is very dynamic so it may happen that your bundle starts before the config admin service or that the config admin service did not yet read the configuration. So you may end up sometimes getting Null for the configuration.

So the recommended way is to use a ManagedService and react on updates. If your bundle can not start without config then it is a good idea to create the pojo class to be configured on the first update received.

## Introducing our very simple class to be configured

As we want to implement a clean style of how to work with configuration the class to be configured should be a pure pojo. While it is of course possible to simply implement the ManagedService interface and work with the Dictionary directly this will make you depend on OSGi and the current Config Admin Service spec. So instead we use a simple bean class that has a title property. Additionally I added a refresh method that should be called after all configuration was changed.

|  |
| --- |
| public class MyApp {    String title;      public void setTitle(String title) {      this.title = title;    }      public void refresh() {      System.out.println("Configuration updated (title=" + title + ")");    }  } |

So our goal is to configure the title when the configuration changes and then call refresh. We will do this in pure OSGi and in blueprint.

## Get some practice. Working with configs using pure OSGi interfaces

The first practical part in this tutorial shows how to use the config admin service using just OSGi interfaces. While this is probably not the way you will do it later it helps to understand what happens under the hood.

You can find the implementation in the subdirectory configapp (<https://github.com/cschneider/Karaf-Tutorial/tree/master/configadmin/configapp>)

So we first need a pom file for the maven build. You best start with the pom from the configapp example.  
If you start fresh you will have to use the maven-bundle-plugin to make your project a OSGi bundle and you need to add two dependencies:

|  |
| --- |
| <dependency>    <groupId>org.osgi</groupId>    <artifactId>org.osgi.compendium</artifactId>    <version>4.2.0</version>  </dependency>  <dependency>    <groupId>org.osgi</groupId>    <artifactId>org.osgi.core</artifactId>    <version>4.2.0</version>  </dependency> |

The first is for the config admin service interfaces and the second is to be able to create the Activator and contains the basic OSGi interfaces.

Now we will care about updating the MyApp class. The following little class does the trick. We implement the ManagedService interface to talk to the Config Admin Service. So we get called whenever the config changes. The first thing is to check for null as this can happen when the config is removed. We could a this point stop our MyApp but to keep it simple we just ignore those. The next step is to create the MyApp class. Normally you would do this in the Activator but then you would have to be able to work with an empty configuration which is not always desired. The last part is to simply call the setter with the value from the config and call refresh after all settings were made.

|  |
| --- |
| private final class ConfigUpdater implements ManagedService {      public void updated(Dictionary config) throws ConfigurationException {      if (config == null) {        return;      }      if (app == null) {        app = new MyApp();      }      app.setTitle((String)config.get("title"));      app.refresh();    }  } |

Of course this does not yet do anything. The last step is to register the ConfigUpdater in the Activator.start. We simply use registerService like for every other service. The only special thing is that you have to set the property SERVICE\_PID to your config pid so the Config Admin Service knows what config you want to watch.

|  |
| --- |
| Hashtable<String, Object> properties = new Hashtable<String, Object>();  properties.put(Constants.SERVICE\_PID, CONFIG\_PID);  serviceReg = context.registerService(ManagedService.class.getName(), new ConfigUpdater() , properties); |

## Making this simple example run

* build the project with mvn install.
* Start a fresh Karaf instance
* Copy the configapp.jar bundle from the target dir to the Karaf deploy dir

Now we notice that nothing seems to happen. Calling list in the Karaf console you should be able to see that the bundle is indeed started but it will not do create any output as there is no config.  
We still need to create the config file and set the title.

* copy the existing file /configadmin-features/src/main/resources/ConfigApp.cfg to the /etc dir of the Karaf instance

The important part here is that the filename has to be <pid>.cfg. So the config admin service will find it.

Now the fileinstall bundle will detect the new file in etc. As the ending is .cfg it will consider it to be a config admin resource and create or update the Config Admin Service configuration with the pid determined from the file name.

So you should now see the following in the Karaf console. This shows that the configuration change was correctly detected and forwarded. If you now change the file using an editor and save the change will alsobe propagated.

|  |
| --- |
| Configuration updated (title=" + title + ") |

## Digging into the config with the Karaf config commands

Type the following in the Karaf console:

|  |
| --- |
| > config:list  Pid:            ConfigApp  BundleLocation: file:/C:/java/apache-karaf-2.2.3/deploy/configapp.jar  Properties:     service.pid = ConfigApp     felix.fileinstall.filename = file:/C:/java/apache-karaf-2.2.3/etc/ConfigApp.cfg     title = my Title |

Among other configs you should find the above config "ConfigApp". The config shows where it has been loaded from, the pid and of course all properties we set in the file.

We can also change the config:

|  |
| --- |
| > config:edit ConfigApp  > config:propset title "A better title"  > config:proplist     service.pid = ConfigApp     felix.fileinstall.filename = file:/C:/java/apache-karaf-2.2.3/etc/ConfigApp.cfg     title = A better title  > config:update  Configuration updated (title=A better title) |

We see that the change is directly propagated to our bundle. If you look into the config file in etc you can see that the change is also persisted to the file. So the change will still be there if we restart Karaf.

## Configuration with Blueprint

After we have worked with the Config Admin Service in pure OSGi we will now look how the same can be achieved in Blueprint. Fortunately this is quite easy as Blueprint does most of the work for us.

We simply define a cm:property-placeholder element. This works similar to property place holder with files but works with the Config Admin Service. We need to supply the config PID and the update strategy.  
As strategy we select "reload". This means that after a change the blueprint context is reloaded to reflect the changes. We also set default properties that will be used when the config PID is not found or an attribute is not present.

The integration with our bean class is mostly a simple bean definition where we define the title property and assign the placeholder which will be resolved using the config admin service. The only special thing is the init-method. This is used to give us the chance to react after all changes were made like in the pure OSGi example.

For bluenprint we do not need any maven dependencies as our Java Code is a pure Java bean. The blueprint context is simply activated by putting it in the OSGI-INF/blueprint directory and by having the blueprint extender loaded. As blueprint is always loaded in Karaf we do not need anything else.

|  |
| --- |
| <blueprint xmlns="[http://www.osgi.org/xmlns/blueprint/v1.0.0"](http://www.osgi.org/xmlns/blueprint/v1.0.0)  xmlns:cm="[http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"](http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0)  xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance)  xsi:schemaLocation="  [http://www.osgi.org/xmlns/blueprint/v1.0.0] [[http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd]](http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd%5d)  [http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0] [[http://svn.apache.org/repos/asf/aries/trunk/blueprint/blueprint-cm/src/main/resources/org/apache/aries/blueprint/compendium/cm/blueprint-cm-1.1.0.xsd]](http://svn.apache.org/repos/asf/aries/trunk/blueprint/blueprint-cm/src/main/resources/org/apache/aries/blueprint/compendium/cm/blueprint-cm-1.1.0.xsd%5d)  ">  <cm:property-placeholder persistent-id="ConfigApp" update-strategy="reload" >  <cm:default-properties>  <cm:property name="title" value="Default Title"/>  </cm:default-properties>  </cm:property-placeholder>    <bean id="myApp" init-method="refresh">  <property name="title" value="$\{title\}"></property>  </bean>  </blueprint> |

In the above xml please remove the backslashes around title. This is just to avoid confluence interpreting it as a wiki macro.

## Deploying config files

After we have successfully used the Config Admin Service the only thing that remains to go into production is to deploy our bundle together with a default configuration. This can be done using a Karaf feature file. We define our feature with the bundles it needs and simply add a configfile element. This makes Karaf deploy the given file into the etc directory of our Karaf installation. If the file is already present then it will not be overwritten.

|  |
| --- |
| <feature name="tutorial-configadmin" version="${pom.version}">  <bundle>mvn:net.lr.tutorial.configadmin/configapp/${pom.version}</bundle>  <bundle>mvn:net.lr.tutorial.configadmin/configapp-blueprint/${pom.version}</bundle>  <configfile finalname="/etc/ConfigApp.cfg">mvn:net.lr.tutorial.configadmin/configadmin-features/${pom.version}/cfg</configfile>  </feature> |

So one last question is how to deploy the config to maven for the configfile element to find it. This happens like for the feature with the build-helper-maven-plugin in Karaf See the pom file for details how to use it.

## Summing it up and a look into the future

During this tutorial we have learned how the Config Admin Service works and how to use it with pure OSGi and blueprint. We have also seen how to build and deploy our projects together with documentation.

While this is already very usefull some small things are missing in my opinion. The first thing is that configfile does not really seem to be consistent with the config admin service. In fact Karaf does not use the config admin service to deploy the file. So what I would like to see is that the also existing config element not only writes the config to the config admin service but also persists it. Fortunately my colleague Jean Baptiste is already working on this. See <https://issues.apache.org/jira/browse/KARAF-888>

The other thing is that for enterprise environments a config admin service with some additional features is needed. One thing is that it should be possible to do configuration on a whole network of servers with a central source for configuration and a nice UI. The other thing is that you would like to not only deploy the default config but also the config the admin really wants for the system. So I  imagine that you should be able to define a deployment plan with bundles and features to install but also with the required configuration changes. If this is done right it will allow good audits of deployment and config changes and will also allow an admin to roll back a change in case something goes wrong. I hope we can provide some of this in one of the next [Talend](http://www.talend.com) ESB EE releases.

**Karaf Support**

Apache Camel is designed to work nicely into [Apache Karaf](http://karaf.apache.org) OSGi container.

It includes:

* Camel features descriptor allowing to easily and quickly install Camel in Karaf.
* Karaf commands allowing you to view, start, stop, get info, about the Camel contexts and routes running in the Karaf instance.  
  **NB: Karaf commands are available starting with Camel 2.8.0**, and the commands was renamed in Camel 2.9.0.

Here are the versions that are compatible.

|  |  |
| --- | --- |
| **Camel Version** | **Karaf Version** |
| 2.2.0 | 1.2.0 |
| 2.3.0 | 1.3.0 |
| 2.4.0 | 1.4.0 |
| 2.5.0 | 2.1.0 |
| 2.7.0 | 2.2.0 |
| 2.7.2 | 2.2.2 |
| 2.8.2 | 2.2.4 |
| 2.9.0 | 2.2.5 |
| 2.10.0 | 2.2.8 |
| 2.11.0 | 2.3.1 |
| 2.12.0 | 2.3.2 |
| 2.13.0 | 2.3.3 |
| 2.14.0 | 2.3.7 |
| 2.15.0 | 2.4.0 |

**Preparing Karaf for Camel**

Camel uses several bundles to provide low level package, such as javax.annotation or javax.xml.bind.

Due to that, we mustn't use the default system package coming from the JDK. It means that we need to "exclude" some packages from system packages in order to use packages provided by tiers bundles.

Camel provides a ready to use config.properties file that you can download:

<http://svn.apache.org/repos/asf/camel/tags/camel-2.7.2/platforms/karaf/features/src/main/resources/config.properties>

This file has to be copied into Karaf etc folder.

Karaf 2.2.5 or newer

Notice that Karaf 2.2.5 now provides an extra configuration file that has been configured to work with Camel and CXF, so its recommended to use this file if you use Karaf 2.2.5 or newer. You can prepare Karaf by copying the etc/jre.properties.cxf and override the existing etc/jre.properties file.

**Install Camel in Karaf**

Assuming that you have a running Karaf instance, you can register the Camel features descriptor:

|  |
| --- |
| karaf@root> features:addurl mvn:org.apache.camel.karaf/apache-camel/2.8.2/xml/features |

From Karaf 2.2.6 onwards this has been simplified as there is a new features:chooseurl command that is pre-setup for Camel

|  |
| --- |
| karaf@root> |
| features:chooseurl camel 2.15.3  features:addurl mvn:org.apache.camel.karaf/apache-camel/2.15.3/xml/features |
| features:install camel 2.15.3  features:install webconsole  The uirl is <http://localhost:8181/system/console> . Log in with karaf/karaf and take some time to see what it has to offer. |

Now, we have all Camel features available:

|  |
| --- |
| karaf@root> features:list|grep -i camel  [uninstalled] [2.8.0          ] camel                         repo-0  [uninstalled] [2.8.0          ] camel-core                    repo-0  [uninstalled] [2.8.0          ] camel-spring                  repo-0  [uninstalled] [2.8.0          ] camel-blueprint               repo-0  [uninstalled] [2.8.0          ] camel-test                    repo-0  [uninstalled] [2.8.0          ] camel-cxf                     repo-0  [uninstalled] [2.8.0          ] camel-cache                   repo-0  [uninstalled] [2.8.0          ] camel-castor                  repo-0  [uninstalled] [2.8.0          ] camel-crypto                  repo-0  [uninstalled] [2.8.0          ] camel-http                    repo-0  [uninstalled] [2.8.0          ] camel-http4                   repo-0  [uninstalled] [2.8.0          ] camel-mina                    repo-0  [uninstalled] [2.8.0          ] camel-jetty                   repo-0  [uninstalled] [2.8.0          ] camel-servlet                 repo-0  [uninstalled] [2.8.0          ] camel-jms                     repo-0  ... |

To install Camel, just install the camel feature:

|  |
| --- |
| karaf@root> features:install camel |

You have to install the Camel features depending of your requirements.

For instance, if you want to use blueprint as Camel DSL, you have to install the camel-blueprint feature:

|  |
| --- |
| karaf@root> features:install camel-blueprint |

If, if your route, you use an endpoint like "stream:out", you have to install the camel-stream feature:

|  |
| --- |
| karaf@root> features:install camel-stream |

**Karaf commands**

When you install the camel feature, new Karaf commands become available automatically.

**Karaf commands in Camel 2.8.x**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **camel:list-contexts** | Lists the camel contexts available in the current Karaf instance |
| **camel:list-routes** | Displays the list of Camel routes available in the current Karaf instance |
| **camel:info-context** | Displays detail information about a given Camel context |
| **camel:start-context** | Starts the given Camel context |
| **camel:stop-context** | Stops the given Camel context |
| **camel:info-route** | Provides detail information about a Camel route |
| **camel:show-route** | Renders the route in XML |
| **camel:start-route** | Starts the given route. |
| **camel:stop-route** | Stops the given route. |

**Karaf commands in Camel 2.9 onwards**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **camel:context-list** | Lists the camel contexts available in the current Karaf instance |
| **camel:context-info** | Displays detail information about a given Camel context |
| **camel:context-start** | Starts the given Camel context |
| **camel:context-stop** | Stops the given Camel context (it becomes unavailable and it can't be started afterwards) |
| **camel:route-list** | Displays the list of Camel routes available in the current Karaf instance |
| **camel:route-info** | Provides detail information about a Camel route |
| **camel:route-show** | Renders the route in XML |
| **camel:route-start** | Starts the given route. From Camel **2.10** onwards you can use \* as wildcard to match multiple routes. |
| **camel:route-stop** | Stops the given route. From Camel **2.10** onwards you can use \* as wildcard to match multiple routes. |
| **camel:route-suspend** | Suspends the given route. From Camel **2.10** onwards you can use \* as wildcard to match multiple routes. |
| **camel:route-resume** | Resumes the given route. From Camel **2.10** onwards you can use \* as wildcard to match multiple routes. |

**Karaf commands in Camel 2.10 onwards**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **camel:endpoint-list** | Lists endpoints fromm all camel contexts available in the current Karaf instance |

**Karaf commands in Camel 2.11 onwards**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **camel:route-profile** | To profile route(s) |
| **camel:route-reset-stats** | To reset performance stats on the given route(s) |

**Karaf commands in Camel 2.12.4 onwards**

|  |  |
| --- | --- |
| **Command** | **Description** |
| **camel:context-suspend** | Suspends the given Camel context |
| **camel:context-resume** | Resumes the given Camel context |

**camel:context-list**

The \***camel:context-list**\* command displays the list of CamelContext available in the current Karaf instance:

|  |
| --- |
| karaf@root> camel:context-list  Name                 Status               Uptime  [66-camel-3        ] [Started           ] [14.559 seconds    ] |

It displays the context name/ID (used in others commands), the current status (started/stopped), the uptime (since when the context has been started).

**camel:route-list**

The \***camel:route-list**\* command displays the list of Camel routes available in the current Karaf instance:

|  |
| --- |
| karaf@root> camel:route-list  [route1              ] |

You can also filter the routes by CamelContext:

|  |
| --- |
| karaf@root> camel:route-list 66-camel-3  [route1              ] |

*Tip: use the TAB key to completion on the CamelContext ID.*

**camel:info-context**

The \***camel:context-info**\* command displays detail information about a given CamelContext:

add the \***--verbose**\* option (following the context name) to also list the endpoints

|  |
| --- |
| karaf@root> camel:context-info 66-camel-3  Camel Context 66-camel-3          Name: 66-camel-3          Version: 2.8.0          Status: Started          Uptime: 1 minute    Advanced          Auto Startup: true          Starting Routes: false          Suspended: false          Tracing: false    Properties    Components          timer          properties          log    Endpoints          timer://test          log://test    Routes          route1    Used Languages |

You can see the current Camel version used by the CamelContext, some context attributes, the components involved in the context, and the endpoints defined.

*Tip: use TAB key for completion on the CamelContext name.*

**camel:context-start**

The \***camel:context-start**\* command starts a given CamelContext:

|  |
| --- |
| karaf@root> camel:context-start 66-camel-3 |

*Tip: use TAB key for completion on the CamelContext name.*

**camel:context-stop**

The \***camel:context-stop**\* command stops a given CamelContext. After stopping the context it becomes unavailable and cannot be started again.

|  |
| --- |
| karaf@root> karaf@root> camel:context-stop 66-camel-3 |

*Tip: use TAB key for completion on the CamelContext name.*

**camel:route-info**

The \***camel:route-info**\* command provides detail information about a Camel route:

|  |
| --- |
| karaf@root> camel:route-info route1  Camel Route route1          Camel Context: 66-camel-3    Properties                  id = route1                  parent = 2e7aacc1    Statistics          Exchanges Total: 98          Exchanges Completed: 98          Exchanges Failed: 0          Min Processing Time: 1ms          Max Processing Time: 2ms          Mean Processing Time: 1ms          Total Processing Time: 134ms          Last Processing Time: 1ms          First Exchange Date: 2011-06-29 07:21:57          Last Exchange Completed Date: 2011-06-29 07:23:34    Definition  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <route id="route1" xmlns="http://camel.apache.org/schema/spring">      <from uri="timer:test"/>      <to uri="log:test" id="to1"/>  </route> |

You can see some statistics (the number of processed exchanges, the processing time, etc) and a XML rendering of your route (whatever DSL used to define the route).

*Tip: use TAB key for completion on the route name.*

**camel:route-show**

The \***camel:route-show**\* command renders the route in XML. It's independent from the DSL used to define the route:

|  |
| --- |
| karaf@root> camel:route-show route1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <route id="route1" xmlns="http://camel.apache.org/schema/spring">      <from uri="timer:test"/>      <to uri="log:test" id="to1"/>  </route> |

*Tip: use TAB key for completion on the route name.*

**camel:route-start**

The \***camel:route-start**\* command starts a Camel route:

|  |
| --- |
| karaf@root> camel:route-start route1 |

*Tip: use TAB key for completion on the route name.*

**camel:route-stop**

The \***camel:route-stop**\* command stops a Camel route:

|  |
| --- |
| karaf@root> camel:route-stop route1 |

**Camel 2.9 or newer:**

**camel:route-suspend**

The \***camel:route-suspend**\* command suspends a Camel route:

|  |
| --- |
| karaf@root> camel:route-suspend route1 |

*Tip: use TAB key for completion on the route name.*

**camel:route-resume**

The \***camel:route-resume**\* command resume a Camel route:

|  |
| --- |
| karaf@root> camel:route-resume route1 |

*Tip: use TAB key for completion on the route name.*

**Camel 2.10 or newer:**

**camel:endpoint-list**

The \***camel:endpoint-list**\* command displays the list of the endpoints available in all camel contexts of the current Karaf instance:

|  |
| --- |
| karaf@root> camel:endpoint-list  camel-id             uri                  Status  [test              ] [timer://test      ] [Started           ]  [test              ] [direct://A        ] [Started           ] |

It displays the context name/ID (used in others commands), the URI of the endpoint and the current status (started/stopped).

**Camel 2.12.4 or newer:**

**camel:context-suspend**

The \***camel:context-suspend**\* command suspends a given CamelContext. It then may be resumed again.

|  |
| --- |
| karaf@root> karaf@root> camel:context-suspend 66-camel-3 |

*Tip: use TAB key for completion on the CamelContext name.*

**camel:context-resume**

The \***camel:context-resume**\* command resumes a given CamelContext. After that it returns to the *started* state.

|  |
| --- |
| karaf@root> karaf@root> camel:context-resume 66-camel-3 |

*Tip: use TAB key for completion on the CamelContext name.*

# 

#### 

#### Downloads

To get started, you’ll want to download the following binary distributions.  I installed them under C:Apache (e.g. C:Apacheapache-karaf-2.2.2).  Note that Karaf does not seem to like having spaces in its install path.

* [Apache Maven](http://maven.apache.org/) (3.0.3)
* [Apache Karaf](http://karaf.apache.org/) (2.2.2)

#### Environment

#### Variables

* Set the “M2\_HOME” environment variable to your Maven install path.  Mine is set to “C:Apacheapache-maven-3.0.3″.
* Set the “M2″ environment variable to your Maven bin path.  Mine is set to “%M2\_HOME%\bin”.
* Set the “KARAF\_HOME” environment variable to your Karaf install path.  Mine is set to “C:Apacheapache-karaf-2.2.2″.
* Add the “M2″ environment variable to your “Path” environment variable (e.g. add “;%M2%” to the end of your “Path” variable).

#### Starting

#### Karaf

Open a new command prompt and enter “cd %KARAF\_HOME%” and then “binkaraf.bat”.  Alternatively you can double-click the “karaf.bat” file in the bin directory from Windows explorer.  You may get a Windows Firewall prompt at this point; allow the operation.

At this point, you should see Karaf starting up.  The initial screen looks like this:

karaf.bat: Ignoring predefined value for KARAF\_HOME

\_\_ \_\_                  \_\_\_\_

/ //\_/\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_/ \_\_/

/ ,<  / \_\_ `/ \_\_\_/ \_\_ `/ /\_

/ /| |/ /\_/ / /  / /\_/ / \_\_/

/\_/ |\_|\_\_,\_/\_/   \_\_,\_/\_/

Apache Karaf (2.2.2)

Hit '<tab>' for a list of available commands

and '[cmd] --help' for help on a specific command.

Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown Karaf.

karaf@root>

I prefer to use an SSH client such as PuTTY to access Karaf instead of the command prompt window as it seems a little bit touchy in Windows.  The following will tell you which port the SSHD is listening on:

karaf@root> sshd

SSH server listening on port 8101

karaf@root>

The default username is “karaf” and the password is also “karaf”.

### Installing

### Camel

The following page describes some of this process: <http://camel.apache.org/karaf.html>.

First of all, the Camel project has prepared a special properties file for Camel to run in Karaf.  This file can be downloaded at <http://svn.apache.org/repos/asf/camel/tags/camel-2.7.2/platforms/karaf/features/src/main/resources/config.properties>.  Save this file as “camel.properties” and place it in your “%KARAF\_HOME%\ext” directory.

Next, edit the “jre.properties” file in “%KARAF\_HOME%\ext”.  Find the entries for “javax.activation” and “javax.annotation” and comment them out as follows:

# Standard package set.  Note that:

#   - javax.transaction\* is exported with a mandatory attribute

jre-1.6=

 javax.accessibility,

# javax.activation,

 javax.activity,

# javax.annotation,

Instead, we’ll be exposed to these packages via geronimo.  Enter the following commands to install geronimo-annotation\_1.1\_spec and geronimo-activation\_1.1\_spec (note that your Bundle ID’s may be different than mine):

karaf@root> osgi:install -s mvn:org.apache.geronimo.specs/geronimo-annotation\_1.1\_spec/1.1-SNAPSHOT

Bundle ID: 69

karaf@root> osgi:install -s mvn:org.apache.geronimo.specs/geronimo-activation\_1.1\_spec/1.2-SNAPSHOT

Bundle ID: 70

Next, enter the following commands to install Camel:

karaf@root> features:addurl mvn:org.apache.camel.karaf/apache-camel/2.8.0/xml/features

karaf@root> features:install camel

karaf@root>

This should install the following Karaf features:

* camel/2.8.0
* camel-core/2.8.0
* camel-spring/2.8.0
* spring/3.0.5.RELEASE
* spring-dm/1.2.1.RELEASE

You can verify that these features were installed by issuing the following command (I’ve filtered the list):

karaf@root> features:list

State         Version           Name                          Repository             Description

[installed  ] [2.8.0          ] camel                         camel-2.8.0

[installed  ] [2.8.0          ] camel-core                    camel-2.8.0

[installed  ] [2.8.0          ] camel-spring                  camel-2.8.0

[installed  ] [3.0.5.RELEASE  ] spring                        karaf-2.2.2

[installed  ] [1.2.1          ] spring-dm                     karaf-2.2.2

### Installing

### ActiveMQ-Camel

The activemq-camel component is distributed with ActiveMQ, not camel, so we will need to add the ActiveMQ repository to the Karaf features.  The following page describes this process: <http://activemq.apache.org/osgi-integration.html>.  Note that ActiveMQ is a dependency, so we will also be installing the activemq feature.

In a nutshell, enter the following commands to install ActiveMQ-Camel:

karaf@root> features:addUrl mvn:org.apache.activemq/activemq-karaf/5.5.0/xml/features

karaf@root> features:install activemq

karaf@root> features:install camel-jms

karaf@root> osgi:install -s mvn:org.apache.activemq/activemq-camel/5.5.0

Bundle ID: 90

Now, you should be good to go!

### Configuring

### Routes

I generally prefer to use Spring XML to configure routes, and it works quite well because Karaf automatically re-loads Camel configurations when changes are made.  Here is an example “camel.xml” file.  It should be placed in the “%KARAF\_HOME%\deploy” folder.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | <beans     xmlns="<http://www.springframework.org/schema/beans>"     xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"     xsi:schemaLocation="<http://camel.apache.org/schema/spring> <http://camel.apache.org/schema/spring/camel-spring.xsd>  <http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans.xsd>">       <camelContext id="camel" xmlns="<http://camel.apache.org/schema/spring>" autoStartup="true">          <route id="example-route" autoStartup="true">              <description>Example Camel Route</description>              <from uri="activemq:example.A"/>              <to uri="activemq:example.B"/>          </route>      </camelContext>        <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">          <property name="brokerURL" value="<tcp://localhost:61616>" />      </bean>        <bean id="pooledConnectionFactory" class="org.apache.activemq.pool.PooledConnectionFactory">          <property name="maxConnections" value="8" />          <property name="maximumActive" value="500" />          <property name="connectionFactory" ref="jmsConnectionFactory" />      </bean>        <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">          <property name="connectionFactory" ref="pooledConnectionFactory"/>          <property name="transacted" value="false"/>          <property name="concurrentConsumers" value="10"/>      </bean>        <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">          <property name="configuration" ref="jmsConfig"/>      </bean>  </beans> |

This config creates a Camel route which consumes messages from the “example.A” queue and publishes them to the “example.B” queue (essentially just moving them).  Note the autoStartup=”true” attributes; when the Camel context(s) are loaded by Karaf, Camel does not automatically start them or their routes.  This attribute tells Camel to automatically start up the context and any routes with the attribute as well, which prevents you from having to use the Karaf console to start each route manually with any configuration changes.

### Conclusion

We started with a fresh installation of Karaf, installed Camel, installed the ActiveMQ component for Camel and setup an example route.  Hopefully this process works as flawlessly for you as it does for me.  If you have any questions or run into any problems, feel free to post in the comments and I’ll try to see if I know what may have caused them.

Shows how to run your camel routes in the OSGi server Apache Karaf. Like for CXF blueprint is used to boot up camel. The tutorial shows three examples - a simple blueprint route, a jms2rest adapter and an order processing example.

**Installing Karaf and making Camel features available**

* Download [Karaf 2.3.1](http://karaf.apache.org/index/community/download.html) and unpack to the file system
* Start bin\karaf.bat or bin/karaf for unix

In Karaf type:

|  |
| --- |
| features:chooseurl camel 2.12.0  features:list |

You should see the camel features

**Getting and building the examples**

You can find the examples for this tutorial on github [Karaf Tutorial - camel](https://github.com/cschneider/Karaf-Tutorial/tree/master/camel).

So either clone the git repo or just download and unpack the zip of it.To build the code do:

|  |
| --- |
| cd camel  mvn clean install |

**Starting simple with a pure blueprint deployment**

Our first example does not even require a java project. In Karaf it is possible to deploy pure blueprint xml files. As camel is well integrated with blueprint you can define a complete camel context with routes in a simple blueprint file.

[simple-camel-blueprint.xml](https://raw.github.com/cschneider/Karaf-Tutorial/master/camel/simple-blueprint/simple-camel-blueprint.xml)

The blueprint xml for a camel context is very similar to the same in spring. Mainly the namespaces are different. Blueprint discovers the dependency on camel so it will automatically require the at least the camel-blueprint feature is installed. The camel components in routes are discovered as OSGi services. So as soon as a camel component is installed using the respective feature it is automatically available for usage in routes.

So to install the above blueprint based camel integration you only have to do the following steps:

|  |
| --- |
| features:install camel-blueprint camel-stream |

Copy simple-camel-blueprint.xml to the deploy folder of karaf. You should now see "Hello Camel" written to the console every 5 seconds.

The blueprint file will be automatically monitored for changes so any changes we do are directly refelcted in Karaf. To try this open the simple-camel-blueprint.xml file from the deploy folder in an editor, change "stream:out" to "log:test" and save. Now the messages on the console should stop and instead you should be able to see "Hello Camel" in the Karaf log file formatted as a normal log line.

**JMS to REST Adapter (jms2rest)**

Icon

This example is not completely standalone. As a prerequisite install the person service example like described in [Karaf Tutorial 4](http://www.liquid-reality.de/display/liquid/2011/12/22/Karaf+Tutorial+Part+4+-+CXF+Services+in+OSGi).

The example shows how to create a bridge from the messaging world to a REST service. It is simple enough that it could be done in a pure blueprint file like the example above. As any bigger integration needs some java code I opted to use a java project for that case.

Like most times we mainly use the maven bundle plugin with defaults and the packaging type bundle to make the project OSGi ready. The camel context is booted up using a blueprint file [blueprint.xml](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/jms2rest/src/main/resources/OSGI-INF/blueprint/blueprint.xml) and the routes are defined in the java class [Jms2RestRoute](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/jms2rest/src/main/java/net/lr/tutorial/karaf/camel/jms2rest/Jms2RestRoute.java).

**Routes**

The first route watches the directory "in" and writes the content of any file placed there to the jms queue "person". It is not strictly necessary but makes it much simpler to test the example by hand.

The seconds route is the real jms2rest adapter. It listens on the jms queue person and expects to get xml content with persons like also used in the PersonService. In the route the id of the person is extracted from the xml and stored in a camel message header. This header is then used to build the rest uri. As a last step the content from the message is sent to the rest uri with a PUT request. So this tells the service to store the person with the given id and data.

**Use of Properties**

Besides the pure route the example shows some more tpyical things you need in camel. So it is a good practice to externalize the url of services we access. Camel uses the Properties component for this task.

This enables us to write {{personServiceUri}} in endpoints or ${properties:personServiceUri} in the simple language.

In a blueprint context the Properties component is automatically aware of injected properties from the config admin service. We use a cm:property-placeholder definition to inject the attributes of the config admin pid "net.lr.tutorial.karaf.cxf.personservice". As there might be no such pid we also define a default value for the personServiceUri so the integration can be deployed without further configuation.

**JMS Component**

We are using the camel jms component in our routes. This is one of the few components that need further configuration to work. We also do this in the blueprint context by defining a JmsComponent and injecting a connection factory into it. In OSGi it is good practice to not define connection factories or data sources directly in the bundle instead we are simply refering to it using a OSGi service reference.

**Deploying and testing the jms2rest Adapter**

Just type the following in Karaf:

|  |
| --- |
| features:chooseurl activemq 5.7.0  features:chooseurl camel 2.12.0  features:install  camel-blueprint camel-jms camel-http  features:install activemq-spring  activemq:create-broker  install -s mvn:net.lr.tutorial.karaf.camel/example-jms2rest/1.0-SNAPSHOT |

This installs the activemq and camel feature files and features in karaf. The activemq:create command creates a broker defintions in the deploy folder. This broker is then automatically started. The broker defintion also publishes an OSGi service for a suitable connection factory. This is then referenced later by our bundle.

As a last step we install our own bundle with the camel route.

Now the route should be visible when typing:

|  |
| --- |
| > camel:route-list  Route Id             Context Name         Status  [file2jms          ] [jms2rest          ] [Started           ]  [personJms2Rest    ] [jms2rest          ] [Started           ] |

Now copy the file [person1.xml](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/jms2rest/src/test/resources/person1.xml) to the folder "in" below the karaf directory. The file should be sent to the queue person by the first route and then sent to the rest service by the second route.

In case the personservice is instaleld you should now see a message like "Update request received for ...". In case it is not installed you should see a 404 in the karaf error when accessing the rest service.

**Order processing example**

The business case in this example is a shop that partly works with external vendors.

We receive an order as an xml file (See: [order1.xml](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/order/src/test/resources/order1.xml)). The order contains a customer element and several item elements. Each item specifies a vendor. This can be either "direct" when we deliver the item ourself or a external vendor name. If the item vendor is "direct" then the item should be exported to a file in a directory with the customer name. All other items are sent out by mail. The mail content should be customizeable. The mail address has to be fetched from a service that maps vendor name to mail address.

**How it works**

This example again uses maven to build, a [blueprint.xml](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/order/src/main/resources/OSGI-INF/blueprint/blueprint.xml) context to boot up camel and a java class [OrderRouteBuilder](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/order/src/main/java/net/lr/tutorial/karaf/camel/order/OrderRouteBuilder.java) for the camel routes. So from an OSGi perspective it works almost the same as the jms2rest example.

The routes are defined in net.lr.tutorial.karaf.camel.order.OrderRouteBuilder. The "order" route listens on the directory "orderin" and expects xml order files to be placed there. The route uses xpath to extract several attributes of the order into message headers. A splitter is used to handle each (/order/item) spearately. Then a content based router is used to handle "direct" items different from others.

In the case of a direct item the recipientlist pattern is used to build the destination folder dynamically using a simple language expression.

|  |
| --- |
| recipientList(simple("file:ordersout/${header.customer}")) |

If the vendor is not "direct" then the route "mailtovendor" is called to create and send a mail to the vendor. Some subject and to address are set using special header names that the mail component understands. The content of the mail is expected in the message body. As the body also should be comfigureable the velocity component is used to fill the [mailtemplate.txt](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/order/src/main/resources/mailtemplate.txt) with values from the headers that were extracted before.

**Deploy into karaf**

The deployment is also very similar to the previous example but a little simpler as we do not need jms. Type the following in karaf

|  |
| --- |
| features:addurl mvn:org.apache.camel.karaf/apache-camel/2.9.0/xml/features  features:install camel-blueprint camel-mail camel-velocity camel-stream  install -s mvn:net.lr.tutorial.karaf.camel/example-order/1.0-SNAPSHOT |

To be able to receive the mail you have to edit the configuration pid. You can either do this by placing a properties file  
into etc/net.lr.tutorial.karaf.cxf.personservice.cfg or editing the config pid using the karaf webconsole. (See [part 2](http://www.liquid-reality.de/display/test/2011/09/23/Karaf+Tutorial+2+-+Working+with+the+Config+Admin+Service) and [part 3](http://www.liquid-reality.de/display/liquid/2011/09/26/Karaf+Tutorial+Part+3+-+Improving+configuration+editing+using+the+OSGI+Metatype+Service+and+the+Felix+Webconsole) of the Karaf Tutorial series).

Basically you have to set these two properties according to your own mail environment.

|  |
| --- |
| mailserver=yourmailserver.com  testVendorEmail=youmail@yourdomain.com |

**Test the order example**

Copy the file [order1.xml](https://github.com/cschneider/Karaf-Tutorial/blob/master/camel/order/src/test/resources/order1.xml) into the folder "ordersin" below the karaf dir.

The Karaf console will show:

|  |
| --- |
| Order from Christian Schneider    Count: 1, Article: Flatscreen TV |

The same should be in a mail in your inbox. At the same time a file should be created in ordersout/Christian Schneider/order1.xml that contains the book item.

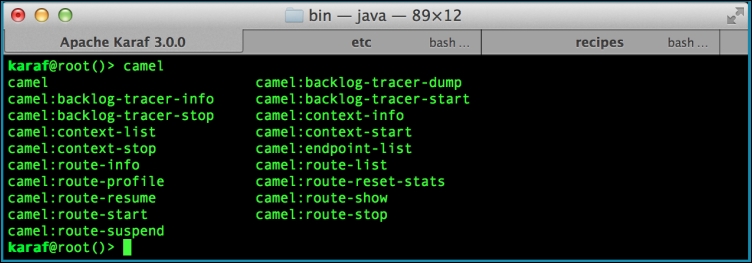
**Wrapping it up and outlook**

The examples show that fairly sophisticated integrations can be done using camel and be nicely deployed in an Apache Karaf container. The examples also show some best practices around configuration management, jms connection factories and templates for customization. The examples should also provide a good starting point for you own integration projects. Many people are a bit hesitant using OSGi in production. I hope these simple examples can show how easy this is in practice. Still problems can arise of course. For that case it is advisable to think about getting a support contract from a vendor like Talend. The whole Talend Integration portfolio is based on Apache Karaf so we are quite experienced in this area.

I have left out one big use case for Apache Camel in this tutorial - Database integrations. This is a big area and warrants a separate tutorial that will soon follow. There I will also explain how to handle DataSources and Connection Factories with drivers that are not already OSGi compliant.

# Listing Camel Contexts in Karaf

The installation of Apache Camel into Apache Karaf includes a set of custom Camel commands as part of the camel-karaf-commands bundle. The Camel community has developed and maintained these commands for the benefit of Karaf users, and as such have helped to fully integrate Camel into the Karaf experience.



Managing Camel Contexts in Karaf is done with two commands, which are as follows:

* camel:context-start contextName: This command is used to start a context
* camel:context-stop contextName: This command is used to stop a context

The following Camel command invocations demonstrate the result of stopping a context:

**karaf@root()> camel:context-list**

**Context Status .**

Starting and stopping the bundle that contains a Camel Context can be very clumsy; alternatively

use the camel:context-start and camel:context-stop commands to manage specific contexts.

# Displaying Camel Context information in Karaf

Karaf can display detailed information about individual Camel Contexts deployed in the container using the camel:context-info command. Context-wide statistics, behaviors, contained components, and more can be discovered using this command.

Use the following camel:context-info command on the Karaf console to retrieve context information—a small warning, there may be a lot of output generated:

**karaf@root()> camel:context-info *CamelCommandContext***

# Listing endpoints in Karaf

Apache Camel users use endpoints to denote URIs from which events and information come from or go to. In Karaf, the endpoint-list command has been provided to help simplify tracking these URIs.

se the camel:endpoint-list command to list all endpoints in Karaf (use the camel:endpoint-list context-name command if you want to restrict output to one context's routes). This is shown in the following command:

# Starting and stopping Camel Contexts in Karaf

Starting and stopping the bundle that contains a Camel Context can be very clumsy; you can use the camel:context-start and camel:context-stop commands to manage specific contexts.

Managing Camel Contexts in Karaf is easy, but requires you to become familiar with two commands, which are as follows:

* camel:context-start contextName: This command is used to start a context
* camel:context-stop contextName: This command is used to stop a context

# Listing endpoints in Karaf

Apache Camel users use endpoints to denote URIs from which events and information come from or go to. In Karaf, the endpoint-list command has been provided to help simplify tracking these URIs.

## Getting ready

Follow the instructions in the Listing Camel Contexts in Karaf recipe's Getting ready section for this recipe.

Use the camel:endpoint-list command to list all endpoints in Karaf (use the camel:endpoint-list context-name command if you want to restrict output to one context's routes).